SoK: Understanding BFT Consensus in the Age of Blockchains

Gang Wang
Email: email.gang.wang@gmail.com

***Abstract*—Blockchain as an enabler to current Internet infrastructure has provided many unique features and revolutionized current distributed systems into a new era. Its decentralization, immutability, and transparency have attracted many applications to adopt the design philosophy of blockchain and customize various replicated solutions. Under the hood of blockchain, consensus protocols play the most important role to achieve distributed replication systems. The distributed system community has extensively studied the technical components of consensus to reach agreement among a group of nodes. Due to trust issues, it is hard to design a resilient system in practical situations because of the existence of various faults. Byzantine fault-tolerant (BFT) state machine replication (SMR) is regarded as an ideal candidate that can tolerate arbitrary faulty behaviors. However, the inherent complexity of BFT consensus protocols and their rapid evolution makes it hard to practically adapt themselves into application domains. There are many excellent Byzantine-based replicated solutions and ideas that have been contributed to improving performance, availability, or resource efficiency. This paper conducts a systematic and comprehensive study on BFT consensus protocols with a specific focus on the blockchain era. We explore both general principles and practical schemes to achieve consensus under Byzantine settings. We then survey, compare, and categorize the state-of-the-art solutions to understand BFT consensus in detail. For each representative protocol, we conduct an in-depth discussion of its most important architectural building blocks as well as the key techniques they used. We aim that this paper can provide system researchers and developers a concrete view of the current design landscape and help them find solutions to concrete problems. Finally, we present several critical challenges and some potential research directions to advance the research on exploring BFT consensus protocols in the age of blockchains.**

I. INTRODUCTION

Blockchain, as a core of many cryptocurrencies, has advanced as a key disruptive innovation with the potential to revolutionize most industries. Its key features of integrity, immutability, transparency, and decentralization have attracted many applications to explore the opportunities on the blockchain. Essentially, blockchain is a replicated, decentralized, trustworthy, and immutable ledger technology, so that data recorded on the blockchain cannot be deleted or modified, and the correctness of data can be verified in a distributed and decentralized manner. It allows a group of participating parties that do not trust each other to provide trustworthy and immutable services without the presence of a trusted intermediary [1]. Blockchain stands in the tradition of distributed protocols for both secure multiparty computation and replicated services for tolerating faults [2]. With blockchain, a group of parties can act as a dependable and trusted third party for maintaining shared states, mediating exchanges, and providing secure computing engines [3]. Consensus is one of the most important problems in any blockchain system, as in any distributed systems where multiple nodes must reach an agreement, even in the presence of faults. Many existing consensus algorithms are mostly applicable to small-scale systems, such as the Practical Byzantine Fault Tolerance protocol (PBFT) [4], which works well with fewer than 20 participating nodes. However, when extending the same protocol to a large-scale scenario, its performance may become unacceptable or even totally unworkable.

G. Wang was with the University of Connecticut, Storrs, CT 06269 USA.

The consensus protocol is the *core* of blockchain to provide agreement services, whose efficiency highly affects the performance and scalability of a blockchain system. Without trusted intermediaries, the parties of blockchain may behave arbitrarily and deviate from the consensus procedures, in which we can literately consider them in a Byzantine environment. Blockchain can benefit from many technologies developed for reaching consensus, replicating state, and broadcasting transactions, but in cases that network connectivity is uncertain, nodes may crash or be subverted by an adversary. Though there are many proof-based consensus protocols for blockchain assisting to solve these issues, e.g., Proof-of-Work (PoW) in Bitcoin [5], they are typically not energy efficient and may cause power starvation. Fortunately, Byzantine fault-tolerant (BFT) state machine replication (SMR) offers some opportunities to design consensus protocols that can tolerate arbitrary faults [6]. Under the hood of BFT SMR, it replicates the state of each replica among the replication system. The capacity to tolerate arbitrary faults makes the BFT replicated system a reality when building some practical and critical applications. However, designing an actual BFT system is not an easy task, due to its inherent complexity.

In general, a consensus protocol must meet three requirements [7]: (a) *Non-triviality.* If a correct entity outputs a value v, then some entity proposed v; (b) *Safety.* If a correct entity outputs a value v, then all correct entities output the same value v; (c) *Liveness.* If all correct entities initiated the protocol, then, eventually, all correct entities output some value. Later, Fisher, Lynch, and Paterson (FLP) [8] proved that a deterministic agreement protocol in an asynchronous network cannot guarantee liveness if one entity may crash, even when links are assumed to be reliable. This is the well-known *FLP impossibility* for asynchronous systems. In an asynchronous system, one cannot distinguish between a crashed node and a correct one. Theoretically, deciding the full network's state and deducing from it an agreed-upon output is impossible. However, there exist some extensions to circumvent the FLP result to achieve an asynchronous consensus, e.g., randomization, timing assumptions, failure detectors, and strong primitives [9]. Over two decades of development, BFT algorithms have evolved into a wide range of protocols and applications. However, these progress were typically designed specifically for some closed groups according to detailed application scenarios.

We distinguish two types of fault-tolerant consensus: crash fault-tolerant consensus (aka. CFT) and non-crash (Byzantine) fault-tolerant consensus (aka. BFT) [10]. Different failure models have been considered in the literature, and they have distinct behaviors. In general, a crash fault is where a machine simply stops all computation and communication, and a non-crash fault is where it acts arbitrarily, but cannot break the cryptographic primitives, such as cryptographic hashes, MACs, message digests, and digital signatures. For instance, in a crash fault model, nodes may fail at any time. When a node fails, it stops processing, sending, or receiving messages. Typically, failed nodes remain silent forever although some distributed protocols have considered node recovery. Tolerating crash faults (e.g., corrupted participating nodes) as well as network faults (e.g., network partitions or asynchrony) reflects the inability of otherwise correct machines to communicate with each other in a timely manner. This reflects how a typical CFT fault affects the system functionalities. Classic CFT and BFT explicitly model *machine* faults only. And these faulty models can be combined with some orthogonal *network* models, e.g., synchronous or asynchronous networks. Thus, the related work can be roughly classified into four categories: synchronous CFT [11], asynchronous CFT [12], synchronous BFT [13], and asynchronous BFT [14] [15]. The Byzantine setting is of relevance to security-critical settings and traditional consensus protocols that tolerate crash failures only.

BFT consensus protocols as the core part of blockchain directly decide if the blockchain technology can be largely applied to practical applications. In the literature, many works discuss various aspects of Byzantine-related protocols, from theory to practical prototype deployment. Although applying BFT protocols to the blockchain is promising, it still faces many design challenges when taking the specific requirements of blockchain into consideration. For example, blockchain typically requires the ordered sequences of transactions in the form of a block including the previous block hash, while some generic BFT algorithms do not need to consider these requirements and only require getting an agreement on the currently processed requests. A systematic-level study on BFT consensus protocols for blockchain is thus highly required. The goal of this paper is to provide a comprehensive survey on the existing Byzantine-related protocols and provide detailed discussions on existing solutions. We aim to provide a concrete view on the state-of-the-art literature in the domain of Byzantine-related consensus and help researchers and system designers find solutions to their specific problems. For each surveyed paper, we try our best to provide detailed information and some potential issues when applying them to blockchain scenarios. Generally speaking, there is some nice literature discussing BFT consensus protocols in some general forms or from architectural/theoretical perspectives. Correia et al. [16] discuss the essential components to achieve consensus under (potential) Byzantine replicas, which focuses on analyzing various Byzantine consensus primarily from a theoretical perspective. Berger and Reiser [17] focus on improving scalability issues for Byzantine consensus, whose application scenarios are for blockchains and distributed ledgers. Bano et al. [18] discuss consensus protocols in general in the age of blockchains, including classic consensus protocols (e.g., Proof-of-X (PoX)) and BFT consensus protocols. Distler [6] provides a complete survey of BFT SMR from a system perspective, by adopting the modulation approach, to discuss each key component in a BFT consensus procedure.

In general, BFT SMR protocols are often challenged for their scalability in terms of replicas, and have not been thoroughly tested for blockchain [19]. From a high-level perspective, most early works focus on the theoretical parts of Byzantine replicated systems, e.g., proof and prototype design, while most recent works focus on the implementation of these prototypes with new feature guarantees, such as paralleling execution and responsiveness. According to current works of literature and their main features, we classify them into different categories. For each category, we present the state-of-the-art literature works in that category and provide some discussion for that category. As a systematization of knowledge on BFT consensus protocols for blockchain, we also provide some research challenges and research directions, which may help interested readers to explore more in the corresponding areas.

The rest of this paper is organized as follows. Section II introduces some preliminary information on Byzantine general problems, BFT algorithms, distributed consensus, and distributed ledger technologies. Section III discusses some commonly used systems models on network's synchrony and adversary ability. Section IV details the existing Byzantine fault-tolerant protocols with clear classification and description. Section V presents the essential components of BFT protocols. Section VI outlines some well-known blockchain consensus algorithms, including PoX and BFT. Section VII discusses some challenges in applying BFT to blockchain. Section VIII provides some discussion and future directions in this domain, and section IX concludes this paper.

II. Preliminaries

This section provides some preliminary information on Byzantine fault tolerance, such as Byzantine generals problem, BFT algorithms, and reliable distributed systems.

A. Byzantine Generals Problem

The Byzantine Generals Problem (BGP) was first introduced by Leslie Lamport, Robert Shostak, and Marshall Pease in 1982 [20]. Practically, it initially was used to handle network communication issues among disconnected units without a coordinator. Originally, it describes a situation where several disconnected Byzantine armies surround an enemy city, and the action of each army is commanded by its own general. The generals can only communicate by messengers with other generals and solely rely on the information from messengers. However, some generals may not work loyally, like traitors (e.g., they try to prevent the loyal generals reaching an agreement), also the messengers do not work in a reliable manner (e.g., with no reliable and authenticated communication channel). Especially, anyone can show the general messages and claim these messages from other generals. A half-baked attack on an enemy city would be a disaster, so all the generals must agree and execute the same plan so that they can gain success. According to the original work [20], in the Byzantine Generals Problem, a commanding general (e.g., one of the generals, working in a leader role) sends an order to his $n-1$ lieutenant generals with the following requirements: (1) All loyal lieutenants obey the same order (e.g., on decision 'retreat' or 'attack'); (2) If the commanding general is loyal, then every individual loyal lieutenant should obey the order sent by the commanding general. The above two requirements together are called the interactive consistency conditions [21]. The problem was shown there to be solvable if and only if fewer than one-third of the generals are faulty - unless unforgeable, signed messages are assumed. In particular, no practical approach works for three generals even with a single traitor.

The Byzantine Generals Problem involves obtaining agreement among a collection of generals, some of which may be faulty. A follow-up work on the *Weak* Byzantine Generals Problem was proposed as a transaction commit problem for a distributed database by Leslie Lamport in 1983 [22]. In a transaction commit problem, there exists a transaction coordinator, and the coordinator's value is its decision of whether to commit or abort the transaction. In a Weak Byzantine Generals (WBG) Problem, it uses the *process* to represent the *general* following the definition in the weak interactive consistency problem. According to the work [22], in a WBG Problem, each process i chooses a private value w_i and the participating processes must then communicate among themselves to allow each process to compute a public value, such that: (1) If all processes are non-faulty and all the w_i have the same value, then every process computes this value as its public value (e.g., final commitment); (2) Any two non-faulty processes compute the same public value. Clearly, any solution to the original Byzantine Generals Problem can be considered as a solution to a WBG Problem, so the WBG Problem is solvable if fewer than one-third of the processes may be faulty.

Briefly, the Byzantine Generals Problem defines the way that the generals handle commands without requiring a centralized coordinator, while still keeping robust even in the presence of malicious generals who may try to betray commands. And, the WBG Problem explains the situations to reach consensus in a system that can have components that give conflicting information because of errors or malicious attacks. Classic consensus algorithms are focused on both types of problems, whose solutions are also called Byzantine-fault tolerant algorithms. Furthermore, these solutions have been somewhat adapted to distributed ledger systems (e.g., blockchain) to reach a consensus among participating parties.

B. BFT Algorithms

Malicious attacks and software failures are becoming more prevalent, which may cause faulty nodes exhibiting Byzantine (i.e., arbitrary) behavior, and the Byzantine fault-tolerant algorithms are thus increasingly important in distributed systems. In a distributed system, replication is a fundamental requirement for implementing dependable services that are able to ensure integrity and availability despite the occurrence of faults and intrusions [23]. In general, a system that has an ability to tolerate Byzantine faults is called a Byzantine tolerant system (for short Byzantine system), in which all non-Byzantine participants (e.g., generals in Byzantine Generals Problem) are required to follow a set of predefined protocols to ensure a consistent state (e.g., consistent decision) among all non-Byzantine participants. State Machine Replication (SMR) [24] [25] is a popular replication method in Byzantine system, which enables a set of replicas (or state machines in SMR) to execute the same sequence of operations for a service even if a fraction of them are faulty. SMR requires an implementation of a total order broadcast protocol, which is known to be equivalent to the consensus problem. Byzantine SMR algorithms typically serve as a standard method for handling Byzantine behaviors, in which they guarantee that all non-Byzantine replicas can together get the same consistent execution result.

Theoretically, many Byzantine fault-tolerant (BFT) total order broadcast protocols provide a solution to the consensus problem and are at the core of any distributed SMR protocols. The basic component for solving SMR in a Byzantine system is the use of BFT reliable broadcast protocol to disseminate the requests among replicas, which ensures that the requests are eventually executed in some consensus instances that define the order of messages to be executed. A Byzantine SMR algorithm typically consists of three sub-protocols: an agreement protocol (aka. consensus protocol), a checkpoint protocol, and a view-change protocol [26]. The agreement protocol typically is used to guarantee the consistency among replicas, e.g., all the requests are committed and executed in a determined order for all non-faulty replicas; the checkpoint protocol periodically checks and clear log information and synchronous status, which typically runs in the background; and the view-change protocol will be activated (e.g., replacing the current ineffective leader in a leader-based protocol) when faults happen during a consensus procedure.

The classic Byzantine consensus in a Byzantine failure mode should follow some properties. Specifically, the Byzantine consensus problem guarantees the conjunction of three properties: (1) *Agreement*, in which no two non-faulty replicas reach different decisions; (2) *Termination*, in which all non-faulty replicas eventually have a decision; and (3) *Validity*, in which the decision is proposed by some replicas. An algorithm fulfilling the above properties can be extended to resolve a Byzantine consensus problem [27]. On the other hand, a BFT SMR algorithm must guarantee the correctness of the following two property [28] [4]: *Safety* and *Liveness*. Safety is the property that if any two non-faulty replicas commit on a decision, then both commit on the same decision, while liveness is to ensure that the consensus protocol makes progress in the current view and moves to a new view, which means clients eventually receive replies to their requests.

The major challenge to safety and liveness in BFT SMR algorithms is the capacity of *equivocation*, i.e., the ability of a Byzantine replica to send inconsistent messages to different non-faulty replicas. Another obstacle to BFT SMR systems is the synchrony issue. Most SMR systems, such as blockchain, operate over a network (i.e., Internet). However, these systems often have some assumptions on network synchrony, in which the delivery of messages must be within a known period of time (details see Section III), and these assumptions might not be realistic in a practical SMR system (based on asynchronous networks). According to Fisher-Lynch-Paterson (FLP) impossibility [29], it is known that, in *asynchronous* networks, consensus is unsolvable even in the presence of a simple crash failure, and needless to say, in Byzantine failures. To deal with this FLP impossibility, many proposed protocols have been designed to relax the guarantees of the classic Byzantine consensus, e.g., Practical Byzantine Fault Tolerant Protocol (PBFT) [4] [27].

PBFT has been long-termly as a consensus protocol to cope with Byzantine systems, which can tolerate up to a $1/3$ fraction of Byzantine faults in a system. We briefly describe its consensus procedures (details can refer to Section IV). One replica, the *primary/leader* replica, decides the order for clients' requests, and forwards them to other replicas, the *secondary* replicas. All replicas together then run a three-phase (pre-prepare/prepare/commit) agreement protocol to agree on the order of requests. Each replica processes every request and sends a response to the corresponding client. The PBFT protocol guarantees that safety is maintained even during periods of timing violations, because progress only depends on the leader. On detecting that the leader replica is faulty through the consensus procedure, the replicas trigger a *view-change* protocol to select a new leader to coordinate the consensus procedure. The leader-based protocol works very well when the number of participating replicas are small, but, it is subject to scalability issues. In general, PBFT is regarded as the baseline for almost all BFT protocols published afterward. Even though many PBFT-like solutions are proposed in the literature, most of them are still subject to scalability issues, which cause them not to fit some large-scale mainstream distributed systems, such as public blockchain systems.

C. Distributed Consensus

The need for reaching agreement among processes which are separated geographically has long been one of the fundamental problems in distributed systems. The distributed consensus problem (and its 'twin' Byzantine Agreement) provide perhaps the most abstract perception for such issues, whose solutions (even in theory) influence many practical designs and implementations [30]. Consensus means that a set of processes reach agreement on a common value by interacting with each other. Reaching such an agreement is a trivial task if both the participants and the underlying communication network are completely trustworthy and reliable. However, such assumptions are not practical in real systems, which often are subject to various faults and attacks, such as service or system crashes, network partitioning, and dropped, malformed, or duplicated messages. We can even consider more Byzantine types of faults, in which faulty processes may behave arbitrarily or a consortium of processes collaboratively behave maliciously [29]. Thus, it is desirable to design a reliable agreement protocol even in the presence of these kinds of faults.

Fault-tolerant consensus problems have been extensively studied in the domain of distributed systems. Even with the existence of various faulty processes and unreliable networks, the fault-tolerant consensus protocol still has the ability to guarantee the requirements of agreement among all non-faulty participants. This is the basic requirement to guarantee the normal and correct functioning of distributed systems [31]. In the literature, there are many fault-tolerant consensus protocols proposed, either for Byzantine or for fail-stop faults (i.e., fail to respond to the requests). In general, there are three basic aspects to evaluate the quality of an algorithm for distributed consensus: resiliency (e.g., the maximum tolerable number of faulty processes), the number of communication rounds, and the maximum message size [30]. Thus, it is desirable to achieve higher quality in these three aspects for a distributed consensus.

Different types of faults (e.g., Byzantines or fail-stop faults) affect the design of a consensus protocol, and the choice of communication models also affects the quality of a consensus protocol. According to different application scenarios, various delicate distributed consensus protocols have been proposed to achieve higher quality by relaxing the conditions of a distributed system. In general, five critical factors affect the design of a distributed consensus as well as its quality, as discussed in [32]: (1) Processors synchronous or asynchronous; (2) Communication synchronous or asynchronous; (3) Message order synchronous or asynchronous; (4) Broadcast transmission or point-to-point transmission; and (5) Atomic receive/send or separate receive and send. The detailed information on synchronous and asynchronous refers to Section III.

As discussed in BFT algorithms, a correct consensus protocol must satisfy at least three conditions: Agreement (e.g., all non-faulty processes choose the same value), Termination (e.g., all non-faulty processes eventually decide), and Validity (e.g., the final output is a value proposed by some process).

D. Distributed Ledger Technologies

Distributed Ledger Technology (DLT) is a general term to describe technologies for the storage, distribution, and exchange of data between users over private or public distributed computer networks, and its database is spread and stored over different locations. Essentially, a distributed ledger is a type of database or entity, shared, replicated, and maintained by the peers of a P2P network. This shared database is available and accessible for all network participants within the system, whose data (aka. transactions in terms of a database) are replicated across multiple storage nodes with equal rights. The action of recording transactions can be considered a result of data exchange among the participants of the network, and only validated results can be appended into the ledger. Also, the distributed ledger can be organized in a decentralized manner, where no trusted third party is required to manage and control the system run, and the participating nodes can automatically reach an agreement via a well-established consensus mechanism. A consensus mechanism is designed to achieve agreement on the respective state of replications of stored data between distributed database nodes under consideration of network failures [33]. DLT in general offers some unique advantages compared with traditional information systems (e.g., databases) and provides some new features and structures to modern applications. Among exiting DLTs, the most well-known data structure is the blockchain. Blockchain-based ledger systems provide some novel features, such as decentralization, trustworthiness, and replicated data synchronized among separated network nodes. In the blockchain, each block associates with a cryptographic hash to its previous block and a timestamp, which makes the ledger immutable and auditable. Any modifications on the transaction inevitably produce an altered hash within its branch, and this alternation is easily detected with little computational effort.

Based on different application scenarios, blockchain can be roughly classified into three categories, namely public (or permissionless), private (or permissioned), and consortium (or federated) blockchain [34] [35] [36] [37].

a) Public Blockchain: A public blockchain is an open and transparent network, which implies that anyone can join and make transactions as well as participate in a consensus process. Also, it is referred to as *permissionless* blockchain, which functions in a completely distributed and decentralized way. The permissionless blockchain makes it possible for anyone to maintain a copy of the blockchain and participate in the validation process of new blocks. Typically, this type of blockchain is adopted by cryptocurrency cases, such as Bitcoin and Ethereum. A permissionless blockchain is typically designed to accommodate a large number of anonymous nodes, so minimizing potential malicious activities is essential. Due to the anonymous participating process, it requires some kind of "proofs" to show the validity of new blocks before publishing them in a public blockchain. For example, proof could be solving the computationally intensive puzzle or staking one's cryptocurrency.

b) Private Blockchain: A private blockchain, on the other hand, is an invitation-only network managed by a central authority[1]. All participants in this blockchain must be permissioned by a validation mechanism to publish or issue transactions. This implies that any node joining a private blockchain is a known and authorized member of a single organization. Typically, a private blockchain is suitable for a single enterprise solution and is used as a distributed synchronized database designed to track information transfers between different departments or individuals. In particular, private blockchain does not need incentive mechanisms (e.g., currencies or tokens) to work, so the transaction processing fee is not needed. Note that the blocks in a private blockchain can be published and agreed on by delegated nodes within the network, hence, its tamper-resistance might not be as effective as a public blockchain.

c) Consortium Blockchain: Consortium blockchain, also known as the federated blockchain, is similar to the settings on a private blockchain, meaning that consortium blockchain requires permission to access the blockchain network. Consortium blockchains, in most cases, cover many parties, which together maintain consistency and transparency among involved parties. Thus, a consortium

[1]This central authority does not participate in blockchain construction, and it mainly provides identification-related services.

blockchain can be considered as a verifiable and reliable communication medium, which is used to trace the shared and synchronized information among its participating members. Similar to the private blockchain, the consortium blockchain also has no transaction processing fees or computational expenses for publishing a new block.

Due to the internal drawbacks of BFT protocols (e.g., scalability), it is a difficult task to apply BFT in a public blockchain system. We typically adopt BFT protocols in private or consortium blockchains to achieve a consensus within the participating nodes.

III. System Models

This section provides the models that a typical BFT consensus may consider during the design. Specially, we focus on the network models and adversary models. And these models defined in this section align with other literature works, e.g., [27], [38], and [39].

A. Network Models

We first explore some potential network connectivity between participants to fulfill the blockchain setting, and then discuss the models of network synchrony. In this paper, we consider the terms *node*, *replica*, *party*, *entity*, and *participant* having the same meaning as *participating node*.

1) Communications: Participants in BFT consensus protocols are required to know the status of their neighbors (like the generals in BGP Problem), whose network layer enables participants to send messages to each other. Specially, in the blockchain era, it often requires a 'full' connectivity. Technically, this 'full' connectivity can be achieved in two manners: point-to-point connectivity, and message "diffusion" (e.g., in a peer-to-peer (P2P) communication setting) [39].

a) Point-to-Point Channel: The point-to-point protocol (PPP) is a basic link transport protocol that can transport packets between two endpoints or parties, whose channel can deliver packets in order [40]. The endpoints are pairwised with reliable and authentic channels. Thus, the point-to-point channel can provide a reliable message transmission. When an endpoint sends a message, it requires to clearly specify its recipient, and the message is typically guaranteed to be received by its recipient as long as there exists a connection between them. The recipient can have the ability to identify the sender as well when receiving a message. And, all parties can know the other parties running the protocol by such a fixed connectivity setting. In some earlier consensus scenarios (e.g., [20]), the full connectivity is considered as the standard communication setting. In PPP, the communication costs are typically measured by the number of messages exchanged during a protocol run. Due to the existence of full connection (as well as communication complexity in the number of messages), the efficiency would be compromised. Thus, it is difficult to be adopted into large-scale consensus protocols, and current popular blockchain systems prefer peer-to-peer diffusion.

b) Peer-to-Peer Diffusion: P2P networks are distributed systems in nature, without any hierarchical organization or centralized control, and peers form self-organizing overlay networks that overlayed on the Internet Protocol (IP) with each peer having symmetry in roles [41]. In a P2P setting, message transmission typically is via a "gossiping" process, i.e., messages received by a peer are transmitted to its neighboring peers (as long as it is not the recipient), whose message transmitting operation can be considered as a kind of messages "diffusion". In general, a P2P network does not provide a reliable message transmission, e.g., no authenticated transmission and no guarantee on the order of messages. If an honest party diffuses a message, it typically does not need to specify a particular recipient, and it can ensure that all its active honest parties get the same message. However, if the sender is compromised, it cannot guarantee that property which means different parties may receive different messages. Thus, P2P diffusion focuses on the message transmission of honest behaviors. The peers have no information about their neighbors, e.g., neither know the identities of their neighbors nor their precise number. And this feature perfectly fits the setting on Byzantine scenarios and the blockchain setting using BFT as its consensus protocol. In P2P, the communication cost depends highly on the underlying network graph, and there are many novel gossiping protocols that achieve $O(n)$ communication complexity, e.g., gossiping protocol based on Information Dispersal Algorithm (IDA) in RapidChain [42]. In general, gossip-based protocols is achieved by the peer-to-peer diffusion, considering that each node has some direct point-to-point connections with other neighboring nodes (e.g., at least a subset of nodes in the network - aka. the degree of the node), and this degree typically is a security parameter to guarantee the connectivity of the underlying network.

Both PPP and P2P models have their advantages and disadvantages. For example, when adopting PPP protocol in a large-scale distributed system, the communication complexity and links may be big issues, and the P2P model does not have a reliable communication guarantee. Besides the above two popular network communication models, there exist some other mixed models considering various real-world scenarios in message passing. For example, an intermediate model by integrating point-to-point channels and diffusion [43]; other intermediate models integrate different criteria (e.g., in terms of partial knowledge of parties [44] and authentication [45]) to improve message passing efficiency.

2) Synchrony: Consensus protocols typically require certain assumptions on the timing information of how messages will be propagated across nodes, which is an important aspect during a protocol design. In BFT protocols, the protocol execution is typically in the form of *rounds*, where each party has the opportunity to send messages for some special event (e.g., a vote on messages) and those messages are supposed to be received by their recipients before the next round. Practically, the concept of round provides a certain level of coordination to guide the protocol execution. And network synchrony is used to define the level of timing coordination among all participating parties. Following the literature work [46] [39] [47], the network synchrony literally can be categorized into three levels, namely *synchronous*, *partially synchronous*, and *asynchronous*.

a) Synchronous: This level of synchrony model is the strongest assumption among the three levels, whose message transmission has a strict time constraint among the parties. Synchronous protocols typically proceed in rounds, and in each round, all processes have a strict requirement on time and the network connectivity is good, so that the message transmitted from one node to its recipient follows the clear time limitation. In general, we say a network is synchronous if its message delivery process (including message processing and message transmission) can be finished within a *fixed* delay (e.g., the delay δ). Typically, this level of synchrony would require a centralized clock synchronization service. We emphasize that the delay δ must be a pre-fixed value between the honest nodes.

b) Asynchronous: This level of asynchrony model is the weakest assumption, where operations of processes are hardly coordinated. An asynchronous network does not provide any guarantee on message delay except that the message will be eventually delivered to its destination. Under an asynchronous setting, the message can be arbitrarily delayed and there is no reliable estimation of this kind of delay. That means, messages between honest nodes can get delivered, however, no upper bound in time can be estimated on their delivery. This kind of asynchrony is typically caused by the lack of clock synchronization service or the existence of the adversary over the communication channels (e.g., delaying or dropping messages). In this kind of network, the message transmission is solely driven by some events (e.g., message delivery events). In a fully asynchronous setting mentioned in [48], the messages may get arbitrarily delayed or simply dropped and the communication is public and unauthenticated. In this kind of setting, the consensus is trivially impossible.

c) Partially Synchronous: This level of synchrony model lies in the middle of synchronous model and asynchronous model. Following the definition in [46], there exist at least two ways to achieve partially synchrony. One possible situation could be that an upper bound δ on message delivery time exists, but we do not know what it is prior; another possible situation is that we know this upper bound δ, but the message processing system may be *unreliable*, causing the message delivery to be late or not at all. The *unreliable* message processing system may work like an asynchronous counterpart, that is, with no time guarantee on when the message will be sent out. Thus, communication is considered as being *partially synchronous* if one of these two situations holds: δ exists but is not known, or δ is known and has to hold from some unknown point on.

According to the FLP impossibility [29], under the asynchronous setting, it has been proven that no deterministic consensus protocol exists even under a single crash failure. However, in practice, the FLP impossibility restriction can get circumvented by using some randomized protocols, e.g., the work [9].

B. Adversary Models

In the context of a consensus protocol, the adversary model (also referred to as the failure model) typically is used to describe the fraction of malicious or faulty nodes that can be tolerated. The literature on consensus protocols typically considers two types of parties -*honest* parties and *corrupted* parties. Typically, honest parties are assumed to behave honestly and strictly follow the pre-defined protocol, also they are assumed to be *online* for the entire execution processes of the protocol [49]. We can roughly categorize the corrupted faults into two classes: fail-stop fault and Byzantine fault.

1) Fail-stop: This kind of fault also refers to the *crash failure* in some literatures. In this model, nodes may get failed at any time, e.g., failing to provide service or response [18] [50]. The failed node typically keeps silent forever (like a “dead" node), and once stopped they cannot restart by themselves. Some common reasons can cause this fault, such as a power shutdown, a software error, and a DoS attack [47]. Under this type of fault, the systems can still continue working correctly if the number of failure nodes is less than half of the total number of participating nodes [51]. There exist some well-known consensus protocols to tolerate a crash failure [38], e.g., Paxos [52] [53], Viewstamped Replication (VSR) [12], ZooKeep [54], and Raft [55].

2) Byzantine: The term of Byzantine failure (aka. Byzantine) came from the Byzantine Generals Problem in distributed systems.

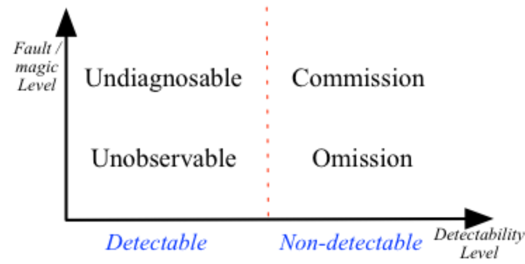


Fig. 1. Abstract classification of Byzantine faults.

Compared with crash failures, a Byzantine failure is a much server, whose process may behave arbitrarily but appears to be a normal one to other nodes. For example, a Byzantine node can send some specific messages to bias the other nodes; also, it can send contradicting messages to different nodes to subvert the consensus. Typically, Byzantine nodes are under the control of the adversary with some malicious intent. Thus, the most common reason for a Byzantine failure is to perform the adversarial influence (actively from either internal or external adversaries), such as a malware injection and a physical device modification. Also, a group of Byzantine processes can collude together to launch a more severe attack, and they may be under the control of one or more adversaries. One of the most known algorithms to handle Byzantine failure is the PBFT, which can achieve consensus in the presence of a certain number of Byzantine nodes. We will discuss the PBFT protocol later in detail.

From different perspectives, a Byzantine failure can be further categorized into different categories. For example, according to message authentication mechanisms, a Byzantine failure can either be an authenticated Byzantine or a general Byzantine [46]. An authenticated Byzantine may behave arbitrarily, but its transmitted messages get correctly signed by the message sender and its signature cannot be forged by any other process. A general Byzantine also behaves arbitrarily and without a signature mechanism, but we typically assume that it is easy to obtain the identity of the message sender.

Similarly, according to the detectability during the process execution, a Byzantine failure can be roughly categorized into *detectable* and *non-detectable* Byzantine faults [56] [57]. We follow the definitions in the literature [56]. The non-detectable Byzantine fault includes two main faults: (1) unobservable faults (i.e., cannot be observed by processes based on the messages they received), and (2) undiagnosable (i.e., cannot be attributed to a particular process). For instance, if a Byzantine process sends to all processes a message claiming that its initial value is something other than its internal input value, then this behavior is unobservable. Alternatively, if a Byzantine process sends a message that purportedly was sent by another process but that does not contain a valid signature, in the absence of further information, then this fault is undiagnosable. The detectable Byzantine faults typically are caused by the external behavior of a process, whose deviations from the specified protocol can be observed by the messages sent (or not sent) by processes during a particular execution of the protocol. This kind of detectable faults can also further be divided into the *omission* faults and *commission* faults. An omission fault occurs when a process fails to send a message that it should have sent in the execution of protocol (e.g., withholding messages at a particular protocol step and no correct process ever received this message), and this behavior can be sensed and observed at the recipient. And, a commission fault happens in the case that a process sends some messages that it is supposed to send during the

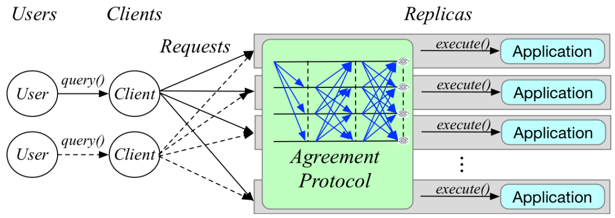


Fig. 2. Abstract of BFT replication system. Users send requests to replicas via client interfaces (with well-defined client library). Replicas together run an agreement protocol to obtain an order on clients' requests, and then each replica executes them in its stateful application [6].

execution of the protocol (e.g., sending unnecessary/vague messages at a particular protocol step, or sending contradictory messages to different correct processes).

Fig. 1 shows an abstract classification of Byzantine faults. The fault level means the malice (i.e., the desire or intention to perform malicious behavior), and typically a higher malice means it is controlled by a stronger adversary. Typically, the Byzantine faults that we discussed in the later sections lie in the domain of detectable Byzantine faults.

IV. Detailed Byzantine Protocols

This section lists some critical Byzantine replicated protocols in detail. We conduct a systematical and comprehensive review of the state-of-the-art protocols to tolerate Byzantine behaviors. Specially, we carefully studied more than one hundred well-known Byzantine-related research works in the range of last 30 years, from PBFT in 1999 to the most recent works in 2021. We tried to group them into distinct categories. However, many protocols involve multiple categories. Instead of categorizing them into distinct categories, we classify them according to these *main* technologies/features/contributions to distinct categories to avoid overlapping discussion in different categories. Each category is not orthogonal, and some of the protocols in them can also be put into other categories. Besides, for each category, we typically follow a *chronological* order (e.g., when the work is published) until the writing of this work. In this whole section, when we say "prior" or "previous" works, it means these works that were published before the time of the current discussed paper; while we use the term "the state-of-the-art" to represent the time of writing this paper.

We note that, to guarantee the originality and integrity of the studied papers in this section, we try to use the same terminologies the authors used and keep these original descriptions on some complex schemes. For example, different papers may utilize different terms, such as "party", "peer", "participant" or "participating node", to represent a replica in a traditional Byzantine replication system.

A. Classic BFT Protocols

From the formal description of Byzantine Generals Problem in 1982 [20], many distributed algorithms and protocols have been proposed to tolerate Byzantine failure with regard to various contexts. In this subsection, we first select and discuss some classic BFT protocols, e.g., PBFT [4], Q/U [58], Zyzzyva [59], and BFT-SMART [60], and these classic protocols provide some initiatives and baselines to design other BFT protocols. For instance, PBFT has inspired numerous BFT consensus protocols with enhanced security and performance, e.g., Quorum/Update (Q/U) [58], Hybrid Quorum (HQ) [61], and Zyzzyva [62]. Many works extend these classic protocols to make some improvements and robustness.

From a high-level perspective, almost all Byzantine fault-tolerant protocols share a basic objective of assigning each client a unique order in the global service history, and executing it in that order [63]. Fig. 2 shows an abstract of BFT replication system. Users send requests to replicas via client interfaces (with a well-defined client library). Replicas together run an agreement protocol to obtain an order on clients' requests, and then each replica executes them in its stateful application [6]. Roughly speaking, there exist two sets of Byzantine fault-tolerant protocols: *agreement-based* protocols and *quorum-based* protocols. The representative protocols for agreement-based and quorum-based scenarios are PBFT and Q/U, respectively. For agreement-based protocols, they first have replicas communicating with each other to agree on a sequence number of a new request and, when agreed, execute that request after have executed all preceding requests in that order. For instance, PBFT engages a three-phase agreement protocol among replicas before a replica executes a request. For quorum-based protocols, they instead restrict their communication to happen only between clients and replicas, as opposed to among replicas; each replica assigns a sequence number to a request and executes it as long as the submitting client appears to have a current picture of the *whole* replica population, otherwise uses some conflict resolution schemes to bring enough replicas up to speed. For instance, Q/U has a one-phase protocol in the fault-free case, however, when faults occur or clients contend to write the same object the protocol has more phases.

In general, the quorum-based protocols require the clients to cache the historical information (e.g., the request sequence) of all (at least "preferred quorum") replicas, putting many burdens on clients, and this typically is not affordable for lightweight clients.

Besides, there exist some hybrid agreement/quorum protocols that share some agreement-based characteristics and some quorum-based features. The speculative protocol Zyzzyva belongs to in this category, in which Zyzzyva has a primary to get an agreement, and when the primary is faulty, it resorts to a quorum-based recovery procedure.

1) PBFT: PBFT is the first practical BFT SMR protocol developed by Castro and Liskov in 1999 [4], which has gained wide recognition for practicality. It can be implemented in Byzantine and partially synchronous settings. The protocol operates in a sequence of views, each coordinated by a leader, which takes the inspiration from Paxos [52]. Paxos is an SMR scheme proposed by Lamport in 1999 that imitates the ancient Paxos part-time parliament. In general, Poxas is designed specially for fault-tolerant consensus protocols while bearing many similarities to Viewstamped Replication (VR) in 1988, which is a server replication scheme for handling server crashes [12] [64]. Different from Paxos, PBFT can be operated in Byzantine and partially synchronous settings, with the help of the leader to coordinate other replicas. In general, PBFT consists of three sub-protocols: 1) Normal operation, 2) Checkpoint, and 3) View-change [47]. In literature, when mentioning PBFT as a consensus protocol, we typically focus on the normal operation protocol, however, the other two sub-protocols are also important.

The normal operation protocol is shown in Algorithm 1 and its graphic communication pattern is shown in Fig. 3. The normal operation is based on*view* and, within each view, the leader orders the requests, and propagates them through a three-step reliable broadcast to the replicas. Replicas are responsible to monitor the behaviors

Algorithm 1 PBFT (Normal-operation)

1: Client sends an operation request to the primary; ▷ *Phase 0: Request*
2: The primary replays this request to replicas via *Pre-prepare* messages; ▷ *Phase 1: Pre-prepare*
3: Replicas record the request and update local states;
4: Replicas send *Prepare* messages to all servers (replicas and the primary); ▷ *Phase 2: Prepare*
5: Once receiving $\geq 2f+1$ *Prepare* messages, every server updates local state and is ready to commit;
6: Servers send *Commit* messages to each other; ▷ *Phase 3: Commit*
7: Once receiving $\geq 2f+1$ *Commit* messages, every server starts to execute the client requests and then updates local states;
8: Every server replies its result to the client; ▷ *Phase 4: Reply*

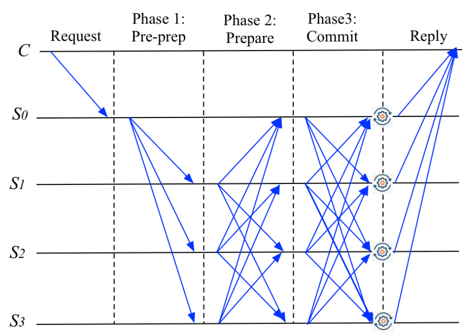


Fig. 3. Message patterns of PBFT

of a leader for both safety and liveness, and can propose a view change request in a case that the leader is unavailable or behaves maliciously [18]. The definitions of safety and liveness can refer to Section II-B. At the end of normal operation, ideally, all results replied to the client should be the same, otherwise, the client chooses the majority result. Typically, the checkpoint protocol is served as a logging tool that keeps a sliding window, whose lower bound is the stable checkpoint, to track active operation requests. The latest stable checkpoint can be used to discard the older request safely from the operation log, as well as facilitate the view change protocol when abnormal cases occur. For instance, in case of a leader failure, the view-change protocol will be triggered by all honest replicas (e.g., detecting a timeout of the leader's message). They think the current leader is compromised and broadcast a view-change message to each other. After receiving more than $2f$ view-change messages from distinct replicas, the next *in-line* (e.g., following round-robin order) replica becomes the newly elected leader and informs the rest to resume the normal operation. In a batch process, the new leader may select a set of different requests for processing, and all non-committed requests in previous normal operations are still in the buffer [65].

In more details on normal operation, PBFT requires $3f+1$ replica to tolerate f faulty replicas. As shown in Fig. 3, a client broadcasts its request to all replicas, and the primary (e.g., S_0) among them assigns a sequence number to the request, and broadcasts this assignment to replicas in a *Pre-prepare* message. A backup replica (e.g., S_1, S_2, and S_3) that receives such assignment acknowledges that and synchronizes with all other replicas on this assignment by broadcasting a *Prepare* message to its neighboring replicas (including itself). When a replica has received $2f+1$ correct *Prepare* messages from distinct replicas, it promises to commit the request at that sequence number by broadcasting a *Commit* message. When a replica has received $2f+1$ such promises for the same request and sequence number, it finally accepts that assignment and executes the request in its local state after it has executed all other requests with lower sequence number, and sends a *Rely* message to the client with the result. A client accepts the result if $f+1$ replicas send the matching *Reply* messages, and otherwise retransmits the request.

The message complexity of PBFT normal operation is $O(n^2)$ to achieve consensus, where n is the number of participating replicas. This quadratic complexity is due to the mutual messaging in both *Prepare* and *Commit* phases, as shown in Fig. 3. The checkpoint protocol typically does not involve any kind of communication complexity. Besides, the view change protocol also requires a quadratic messaging complexity, as it needs to ensure an agreement on the new leader and view, as well as guarantee the safety of messages agreed in previous views. In case of fault tolerance, the properties (e.g., safety and liveness) of PBFT protocol can be guaranteed if $n >= 3f+1$, where f is the number of Byzantine replicas. A short answer for this conclusion is that a replica needs to receive more than $2f+1$ *Prepare/Commit* messages in Prepare/Commit phases respectively before proceeding to the next action, and this will require at least $2f+1$ honest servers in the same state after Commit phase and producing the same result so that the f Byzantine replicas are not able to sway the majority consensus. For detailed proofs on this conclusion, interested readers can refer to the original PBFT [4].

PBFT protocol has been perceived to be a communication-heavy protocol (e.g., by employing replication to achieve resilience against Byzantine failures). This results in PBFT protocol not scalable. And, considering the performance, it is hard to apply PBFT to large-scale distributed systems. Also, it was shown that PBFT can be attacked by an adversary using some specific scheduling mechanism, halting the consensus (e.g., forcing to wait a long timeout when the leader is partitioned and unsynchronized) [66]. However, PBFT, as the first BFT SMR protocol, has indeed inspired numerous BFT consensus protocols with enhanced security and performance, e.g., QU/HQ, Zyzzyva, and HoneyBadger [66]. We will discuss these protocols in the following parts.

2) Q/U: The Query/Update (Q/U) protocol is an optimistic quorum-based protocol enabling fault-scalable Byzantine fault-tolerant services, which was presented by Micheal et al. in 2005 [58]. The optimistic quorum-based nature allows it to provide better throughput and fault-scalability than replicated state machines using agreement-based protocols. A *fault-scalable* service is the one in which performance degrades gradually, if at all, as more server faults are tolerated. And the experience shows that, for Byzantine fault tolerance scenarios, the agreement-based approach is not fault-scalable. This is because in an agreement-based protocol (e.g., [4], [67], [68], [69], [70], [71]), every server processes each request and performs server-to-server broadcast; while in a quorum-based protocol (e.g., [72], [73], [74], [75], [76], [77]), only quorums (or subsets) of servers process each request and the server-to-server communication is generally avoided. Services built with the Q/U protocol are fault-scalable, and it provides an operations-based interface that allows services to be built in a similar manner to replicated state machines. Q/U objects exporting interfaces consists of deterministic methods: *queries* that do not modify objects, and *updates* that do. Compared with previous BFT protocols, the Q/U protocol does not have the concept of the commit phase (not even a lazy commit).

In general, the Q/U protocol requires $5f+1$ servers to tolerate

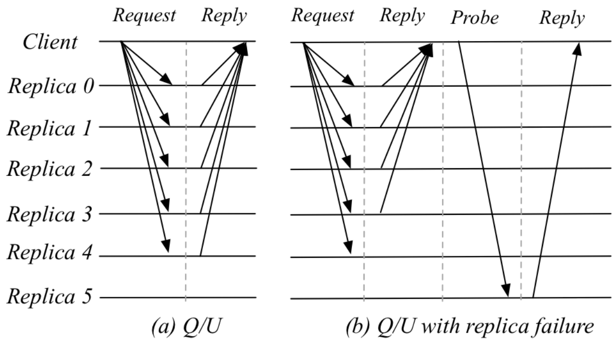


Fig. 4. Message patterns of Q/U. (a) Q/U common case operation, (b) Q/U with replica failure.

f Byzantine faulty servers, while most agreement-based approaches only require $3f + 1$ servers. Even with this imperfection on fault-tolerance rate, the Q/U protocol still achieves its better performance and fault-scalability. This is benefited from the combination of several techniques, i.e., optimism, quorum, and cryptography. The optimism is enabled by the use of non-destructive updates at the versioning service, which permits operations to efficiently resolve contention and/or failed updates. By combining versioning and local timestamping scheme at each update operation, it is impossible for a faulty client to submit different updates at the same timestamp, as these updates intrinsically have different timestamps. The Q/U protocol applies a strategy for accessing the quorums using a *preferred quorum* per object so as to make server-to-server communication an exceptional case. For example, clients initially send requests to an object's preferred quorum and do not issue queries prior to updates for objects. Besides, it employs efficient cryptographic technologies (e.g., *authenticators* in Q/U) to optimize the operation cost. The efficiency of the Q/U protocol is a combinational result of the above three techniques.

In more details on execution, Q/U is a single-phase quorum-based protocol that tolerates up to f faulty replicas in a system of $5f+1$. As shown in Fig. 4, clients cache replicas' histories (the request sequence known by a replica), which they include in requests to replicas, and which they update using replies from replicas. These histories allow a replica that receives a client request to optimistically execute the request immediately, as long as its request history is reflected in a client's view. When a client receives replies, e.g., $4f + 1$ replies indicating they optimistically executed their requests, the request completes. Normally, a client only contacts its "preferred quorum" of replicas instead of the whole replicas set. If some quorum replicas are slow to respond, a client might engage more replicas in via a "probe" hoping to complete the quorum. If the number of replies that accept the request is between $2f + 1$ and $4f$, even others refuse the request, e.g., due to a stale replica history, the client infers there exists a concurrent request from another client. In Q/U's design, it provides a recovery scheme to resolve the conflict in which clients back off and later resubmit requests, after repairing the replicas to make them consistent. Fig. 4(a) shows the best case for a client's request, and Fig. 4(b) illustrates the probing mechanism.

The Q/U protocol is assumed under an asynchronous communication timing model, and both clients and servers may be Byzantine. And the adopted server failure model is a hybrid failure model that combines Byzantine failures with crash-recovery failures. And the rate of the number of Byzantine servers over the total number of hybrid failure servers affects the result. The authors implemented a

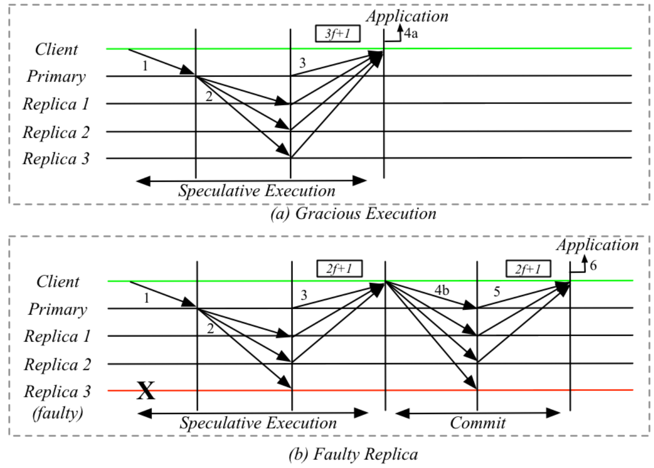


Fig. 5. Protocol communication patterns within a view for (a) gracious execution and (b) faulty replica cases. The numbers refer to the main steps of the protocol numbered in the text [59].

prototype library to demonstrate the efficiency of the Q/U protocol. Overall, the performance of the Q/U protocol decreases by only 36% as the number of Byzantine faults tolerated increases from one to five, whereas the performance of the replicated state machine decreases by 83%.

However, the Q/U protocol has two shortcomings that prevent the full benefit of a quorum-based system. 1) It requires a large number of replicas, e.g., $5f + 1$ to tolerate f failures, which is much higher than the theoretical minimum of $3f + 1$ for agreement-based protocols. This potentially increases in the replica set size. 2). Q/U performs poorly when there are contentions among concurrent write operations. It resorts to exponential back-off to resolve these connections, highly reduced the overall throughput, and its performance would be worse when many applications concurrently perform the write operations [61].

3) Zyzzyva: Zyzzyva [59] [62] is a speculative Byzantine fault-tolerant protocol that was proposed by Kotla in 2007, which uses the speculation technology to reduce the cost of BFT replication system. The concept of speculation is that replicas speculatively execute requests without running an expensive agreement protocol to definitively establish the order as in PBFT. The main idea behind this is that most replicas in a distributed system are normal (e.g., no fault) in most cases, and there is no need to execute the expensive commit protocol for every request. It utilizes fast track and actively involves the client in consensus process. So, in Zyzzyva, replicas reply to a client's request without first running a three-phase BFT like consensus protocol. Replica speculatively receives the requested order from the primary and responds to the client immediately. The client then waits for replies from all peers. In the best-case scenario, if the client receives $3f+1$ replies, it then commits the request. In general, Zyzzyva potentially improves system performance (e.g., via high throughput pipeline of SMR) without too many Byzantine replicas, however, with more Byzantine replicas involved, its consensus efficiency will be compromised. For instance, in the case when the client receives between $2f + 1$ and $3f$ replies, a regular consensus algorithm (e.g., non-speculative slow consensus) is used, and this speculative process definitely decreases system performance [78]. Different from PBFT which does not involve the client in consensus process, the client in Zyzzyva is a key player, who is responsible for checking the integrity of replicas. Fig. 5 shows an abstract communication pattern of Zyzzyva scheme.

Speculation technology does improve BFT's performance, and Zyzzyva is an efficient and scalable consensus protocol for its speculative path. However, this efficiency comes at a price, in the form of fragility, which is mainly due to its two commit paths [79]. One path is the speculative path (aka. fast path), and the other path is a two-phase path that resembles PBFT (aka. slow path). The fast path does not have commit messages, and a client commits a decision by seeing $3f + 1$ prepare messages. The optimistic mode is coupled with a recover mode that guarantees the progress in face of failure, and this recovery mode typically intertwines the PBFT two-phase steps [80]. For example, if the speculative execution fails and the client must execute its non-speculative fallback, this requires at least two additional rounds of communication in addition to the time spent waiting for the timeout. This definitely negates its key feature of high performance. When applying this to a fast response system (e.g., blockchain system), it may come out two decision tracks of the protocol (via fast and slow paths respectively). In the fast track, a possible decision value manifests itself as $f + 1$ prepare messages; while in the slow track, it manifests itself as a commit-certificate (as in PBFT). This scenario may lead Zyzzyva to break safety, as showed in [80]. Besides, the view-change protocol of Zyzzyva fails to provide safety against a faulty leader.

In more detail, Zyzzyva requires $3f + 1$ replicas to tolerate f faults. In general, Zyzzyva runs in a hybrid model of agreement/quorum protocol. Like PBFT, it uses a primary to order requests. Clients send the request only to the primary. Once the primary has ordered these requests, it submits the ordered requests in an OrderReq message to replicas, which respond to the client immediately as in Q/U. In fault-free and synchronous execution, in which all $3f + 1$ replicas return the same response to the client, Zyzzyva is efficient since requests complete in three message delays, and unlike Q/U, *write-contention* by multiple clients is mitigated by the primary's ordering of requests. When some replicas are slower or faulty and the client receives between $2f + 1$ and $3f$ matching responses, it must marshal those responses and re-send them to the replicas (including the primary), to convince them that a quorum of at least $2f + 1$ has chosen the same ordering for the request. If it receives $2f + 1$ matching responses to this second phase, it completes the request in five message delays. However, Zyzzyva's view change protocol is more heavy-weight and complex than a PBFT, according to the results shown in [59], even one replica is faulty, e.g., muteness, Zyzzyva is slower than PBFT.

In general, Zyzzyva uses fast and speculative scheme to drive its latency near the optimal for an agreement protocol. As in the family of speculative BFT protocols, it also has several variants, e.g., Zyzzyva5 [59] [62], MinZyzzyva [81], AZyzzyva [14] [82], and SACZyzzyva. Here we only provide a brief description of these variants, and detailed information on these variants will be provided in the following sections. Zyzzyva5 is introduced in the same paper as Zyzzyva, which is used to avoid the non-speculative fallback at the expense of fault-tolerant rate by relaxing the number of replicas from $3f + 1$ to $5f + 1$. In protocol, it allows the client to complete requests after receiving $4f + 1$ replies. However, this comes with a lower fault-tolerant rate to $\lfloor (n - 1)/5 \rfloor$ compared with Zyzzyva's $\lfloor (n - 1)/3 \rfloor$. AZyzzyva is a protocol to mimic the behaviors of Zyzzyva in best-case situations, by combing the speculative (fast) path of Zyzzyva (called *Zlight* in the protocol) with a recovery protocol, e.g., PBFT. In case that Zlight fails to make progress, AZyzzyva switches to a *new* view that executes PBFT for a fixed number of log slots. In this way, it can easily be extended to a pipeline of state-machine commands without being vulnerable to the safety violation of Zyzzyva exposed [80]. MinZyzzyva is also based on speculation, which improves Zyzzyva with the help of a trusted monotonic counter in each replica. And this trusted counter can guarantee integrity by the underlying hardware. MinZyzzyva allows a reduction of the number of replicas of Zyzzyva from $3f + 1$ to $2f + 1$, and preserves the same safety and liveness properties. SACZyzzyva is a Single-Active Counter Zyzzyva, by addressing some concerns on resilience to slow replicas. It requires the number of $3f + 1$ replicas to tolerate f faulty replicas with *only* one replica needing an active monotonic counter at any given time, compared with each replica equipped with an active monotonic counter in MinZyzzyva.

TABLE I. COMPARISON OF SPECULATIVE BFT PROTOCOLS TOLERATING f FAULTS, WHERE RESILIENCE REFERS TO THE MAXIMUM NUMBER OF REPLICAS THAT CAN BE NON-RESPONSIVE WITHOUT FALLING BACK TO NON-SPECULATIVE OPERATION [79].

Protocol	*Total # of Replicas*	*Resilience*	*Monotonic Total*	*Counter Active*
Zyzzyva	*3f+1*	*0*	-	-
Zyzzyva5	*5f+1*	*f*	-	-
MinZyzzyva	*2f+1*	*0*	*2f+1*	*2f+1*
AZyzzyva	*3f+1*	-	-	-
SACZyzzyva	*3f+1*	*f*	*f+1*	*1*

Table I shows a comparison of the above speculative BFT protocols, where f is the number of faulty replica the system can tolerate.

4) BFT-SMART: BFT-SMART is an open-source library that implements a modular SMR protocol to tolerate Byzantine faults in some replicas, whose work was started around 2010 [60]. It features the increased reliability and modularity, as well as flexible programming interfaces. The original open-source library was developed in Java and implements a protocol for SMR similar to other Byzantine fault-tolerant protocols, e.g., PBFT, and now it can be extended to different versions on other different programming languages, which will improve the diversity of BFT-SMART for various applications [83]. As an SMR, BFT-SMART provides several benefits, from high-level perspectives: (1) It is highly capable for modern hardware, e.g., multi-core systems; (2) It comes with high performance, compared with other protocols, e.g., PBFT and UpRight, in terms of consensus time (i.e., the time to process a client's request); (3) It can achieve high accuracy in replicated data when a Byzantine faulty behavior is exhibited; and (4) It supports reconfiguration of replicas sets, e.g., addition and removal nodes [84] [85]

Following some existing literatures [83] [86] [87], we describe the working principle of BFT-SMART and its features for practical deployment as an SMR. Fig. 6 shows an abstract communication pattern of BFT-SMART, and detailed information refers to the work [60].

a) SMR Consensus Protocol: BFT-SMART utilizes the Mod-SMART protocol, as a total order multicast service, to implement the SMR functionalities. Mod-SMART functions as a modular SMR protocol that works by executing a sequence of consensus instances based on the leader-based Byzantine consensus algorithms (e.g., PBFT). During its normal operation, the communication pattern is similar to PBFT protocols, where clients send their requests to all replicas and wait for the replies. Following PBFT, each consensus

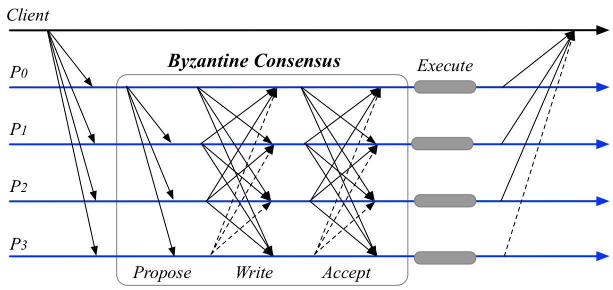


Fig. 6. Message patterns of BFT-Smart [60].

instance begins with a leader proposing a *batch* of client operations to be decided. Specially, each instance consists of three communication steps whose message pattern is shown in Fig. 6. The first step requires the consensus's leader to send a batched *PROPOSE* message to all other replicas, which is followed by two rounds of *all-to-all* communications consisting of *WRITE* and *ACCEPT* messages respectively. In general, *PROPOSE* messages contain a batch of requests to be decided, while both *WRITE* and *ACCEPT* messages only contain the cryptographic hash of such batch to reduce the communication cost.

The above operations are the Mod-SMART *normal phase* execution, which take place in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, e.g., the leader is faulty and/or the network experiences a period of asynchrony, Mod-SMART may trigger a *synchronization phase* to elect a new leader for consensus instances and synchronize all correct replicas. In this phase, BFT-SMART may make some replicas trigger the state transfer protocol.

b) State Transfer: As an efficient and practical SMR, BFT-SMART allows crashed replicas to recover and resume execution without restarting the whole replicated service. To achieve this, BFT-SMART utilizes an intermediate layer between the Mod-SMART protocol and the replicated service, which is responsible for triggering service checkpoints and managing the request log. It thus requires the use of a stable storage to recover this whole system. Typically, there are two ways to achieve state transfer. One utilizes a similar approach in PBFT, in which it requires storing the request log in memory which is periodically truncated after a snapshot of the service state is created. And this is provided by the checkpoint sub-protocol of PBFT. The other way to achieve state transfer is by implementing the efficient durability layers described in [84]. When this layer is enabled, BFT-SMART stores the request log into a stable storage to preserve the service state even if all replicas fail by crashing. The key ideas behind this layer include: 1) logging batches of operations in a single disk while these operations are being executed by the service, 2) taking snapshots at different points of the execution in different replicas to avoid stopping the system, and 3) performing state transfer in a collaborative way, with each replica sending different parts of the state to recover replica. To make the task of writing requests to disk efficiently, these requests can be written to the durable log in parallel; and to utilize the bandwidth efficiently, the system can try to write multiple batches at once.

In general, the durability layer under state transfer enables replicas to execute checkpoints at different moments of their execution and a collaborative state transfer. And, the state transfer service is an independent model, without influencing the consensus protocol.

c) Configuration: BFT-SMART provides a certain mechanism to enable the features of adding and removing replicas from the system *on-the-fly*, reconfiguring the replica set (e.g., the information on view). Such a process assumes the existence of a distinguished trusted client, known as the *View Manager*, which has the ability to issue updates on the replica set by using the aforementioned SMR information. To reconfigure the current replica set, the View Manager issues a special type of operation which is submitted to the Mod-SMART algorithm just like any other client operation, to notify the system that it wants to add replicas to (or remove replicas from) the system. Since these special operations are totally ordered, all correct replicas will adopt the same view as the system's current view at any given point in the execution of client operations. Besides, these special operation requests are never delivered to applications, and the installation is only used to update the view.

After the View Manager receives confirmation from the current replicas that the update is executed (with the verification on the request signature), it notifies the joining replicas that they can start participating in the replication protocol, or the removing replicas will not receive messages from the current replica set. The newly joined replicas then invoke the state transfer protocol to retrieve the latest application state from other replicas before actively participating in the replication protocol. After these replicas successfully update their state, they can start to process new requests.

The above discussion gives a brief overview of the BFT-SMART protocol. In fact, BFT-SMART was the *rare* known project that was developed before the permissioned blockchains surged around 2015 [88]. And there is widespread agreement today that BFT-SMART provides extremely advanced and widely tested implementation of a BFT consensus protocol available [51]. For example, several well-known blockchain projects adopt the BFT-SMART as their consensus protocol, such as Hyperledger Fabric (V1) [89], Symbiont [90], and R3 Corda [91]. Besides, there exist some variants on BFT-SMART, e.g., WHEAT optimized for geo-replicated environment [86], and some visualization services for BFT-SMART, e.g., the work [92].

B. Leaderless BFT Protocols

Most classic BFT protocols in Section IV-A are based on the correct leaders or coordinators to advance or terminate the progress of consensus. For example, protocols run in a series of views, and each view has a delegated node called leader to coordinate all consensus decisions [93]. In general, leader-based BFT protocols rely on the process of view synchronization with the help of leaders. And in "common case" (e.g., no faulty leaders), the leader-based consensus protocols work well and achieve required features. However, when the leader behaves maliciously or goes wrong, it may go through very complex view synchronization to get a correct view to continue. These consensus protocols are mainly based on an ad-hoc way to rotating leaders and synchronize nodes to the same view. However, many literatures [94] [95] [66] [96] have demonstrated that the view synchronization process is complicated and bug-prone. For example, traditional view synchronization relies on the all-to-all communication to ensure all correct nodes reaching the same decision with absolute certainty, which involves a quadratic communication overhead. This would make the protocol difficult to scale to a large number of participants [97]. Besides, the leader may be very slow but correct, which prevents the whole consensus process, and the centralized leader makes the process vulnerable to numerous attacks. Thus, leaderless consensus protocols would help to achieve more features in some applications.

Typically, leader-based consensus protocols would require some assumptions on network synchrony (e.g., time interval for each view or operation). With the wide adoption of blockchain into real-world applications (e.g., via Internet connection), more and more practical blockchain applications require asynchronous. The main idea of asynchronous rounds is that it allows the replicas to commit the request as long as they receive threshold messages, instead of having to wait for a message from a leader/coordinator that may be slow. Leaderless consensus protocols would not require this kind of coordination, in which each replica plays a similar role in the execution of the consensus and make the decision inherently “democratic". Also, without the leader, the protocol works in a decentralization manner, which can avoid bottlenecks by balancing the load and make the consensus more scalable.

From a high-level perspective, a leaderless consensus protocol requires an efficient gossip algorithm and some primitives (e.g., atomic broadcast protocols) to achieve the consensus without relying on some classic coordinators. Different leaderless protocols may have different primitives to achieve their consensus process among replicas. This section presents some well-known leaderless BFT consensus protocols.

1) DBFT: Democratic Byzantine fault tolerance (DBFT) is a leaderless Byzantine consensus for blockchain, proposed by Crain et al. in 2018 [98]. DBFT allows processes to complete asynchronous rounds as soon as they receive a threshold of messages, instead of having to wait for a message from a coordinator that may be slow. DBFT is appealing for blockchains for two reasons: 1) each node plays a similar role in the execution of the consensus, hence making the decision inherently “democratic"; 2) decentralization avoids bottlenecks by balancing the load, making the solution scalable. Essentially, DBFT is deterministic, assumes partial synchrony, is resilience optimal, time-optimal, and does not need signatures. Also, the authors present a simple safe binary Byzantine consensus algorithm, modify it to ensure termination, and finally present an optimized reduction from the multivalue consensus to a binary consensus whose fast path terminates in four message delays.

In general, leader-based Byzantine consensus protocols typically take advantages in case that the leader is non-faulty and the messages are delivered in a timely manner even in an asynchronous round. However, a faulty coordinator can dramatically impact the algorithm performance. DBFT can be used to solve this issue by not relying on a classic coordinator or leader. Instead, DBFT uses a *weak coordinator* that does not impose its value. This provides several benefits: 1) it allows non-faulty processes to decide a value quickly without the help of the coordinator; 2) the coordinator helps the algorithm terminating if non-faulty processes know that they propose values that might all be decided; 3) a weak coordinator allows rounds to be executed optimistically without waiting for a specific message (e.g., from the coordinator in traditional scenarios). In more detail, DBFT assumes an asynchronous process and a reliable asynchronous communication network. Asynchronous processes mean that each process proceeds at its own speed, which can vary with time and remains unknown to the other processes, while a reliable asynchronous network means that there is no bound on message transferring delays but these delays are finite. Based on the above assumptions, the authors propose a binary Byzantine consensus (BBC) algorithm, which relies on binary value all-to-all communication abstraction, denoted BV-broadcast, originally introduced for randomized consensus [99]. The authors also present a safe asynchronous BBC algorithm, which consists of three phases. That means, during a round, each non-faulty process proceeds in three phases: Phase 1 is to broadcast binary value to filter out the values of Byzantine processes; Phase 2 is used to exchange the estimates to coverage to an agreement; and Phase 3 is to decide upon estimate convergence to round number modulo 2 (as a binary scheme). Interested readers can refer to the work [98].

2) EZBFT: EZBFT is a leaderless distributed Byzantine fault-tolerant consensus protocol to minimize the client-side latency in WAN deployments, proposed by Arun et al. in 2019 [100]. The authors claim that, to achieve minimize client-side latency, EZBFT (i) has no designated primary replica, and instead, enables every replica to order the requests that it receives from clients; (ii) uses only three communication steps to order requests in the common case; (iii) involves client actively in the consensus process. Also, EZBFT minimizes the potentially negative effect of a Byzantine replica on the overall system performance. Many previous protocols do not reduce client-side latencies in a geo-scale setting, where the latency per communication step is as important as the number of communication steps. For example, a distant client will incur high latency for the first communication step of sending the request to the primary. While leaderless protocols can potentially resolve this problem, in which a client sends its requests to the nearest replicas and continues to do so as long as the replica is correct. This does not require the primary being involved. To enable leaderless operations, EZBFT exploits a particular characteristic of client commands: interference. In the absence of concurrent interfering commands, EZBFT’s client receives a reply in an optimal three communication steps. However, when commands interfere, both clients and replicas coherently communicate to establish a consistent total order, consuming an additional zero or two communication steps.

In more details, EZBFT requires at least $3f+1$ replicas to tolerate f faulty replicas. EZBFT uses two kinds of quorums: a fast quorum with $3f+1$ replicas and a slow quorum with $2f+1$ replicas. Safety is guaranteed as long as only up to f replicas fail, while liveness is guaranteed during periods of synchronous communication. In general, EZBFT can deliver decisions in three communication steps from the client’s point-of-view, if there is no contention, no Byzantine failures, and synchronous communication between replicas. However, this is an ideal condition. According to the authors, these three communication steps include: (i) a client sending a request to any one of the replicas (closest preferably); (ii) a replica forwarding the request to other replicas with a proposed order; and (iii) other replicas speculatively executing the requests as per the proposed order and replying back to the client. These three steps constitute EZBFT’s core novelty. In general, EZBFT puts many burdens on final decisions to the clients, this assumption does not suit well for lightweight clients.

In EZBFT, it includes many assumptions which help the design of a leaderless consensus protocol, and some of these assumptions are not practical. For example, as the work [101] pointed out, “EZBFT violates the safety property of a consensus protocol. EZBFT also violates liveness and execution consistency; a requirement for total ordering of non-commutative commands." The work [101] provides some enhancements to achieve the original EZBFT claimed properties. Interested readers can read the work [101].

3) Aleph: Aleph is an atomic broadcast protocol in asynchronous networks with Byzantine nodes, proposed by Gkagol et al. in 2019 [102]. Its deprecated version [103] is presented by Gkagol and

Swietek in 2018 targeting for leaderless protocol. From a high-level perspective, Aleph is based on the design of Asynchronous Byzantine Fault Tolerant (ABFT) systems, which allows formally reasoning about correctness, efficiency, and security in the strictest possible model. Also, it makes some improvements on the HoneyBadger BFT by reducing the asymptotic latency while matching the optimal communication complexity. Aleph does not require a trusted dealer with the help of a trustless ABFT randomness beacon. In general, ABFT models are resistant to some harsh network conditions, e.g., arbitrarily long delays on messages, node crashes, or even multiple nodes colluding in order to break the system. However, ABFT models are often considered purely theoretical due to the heavy mathematical formalism. The ABFT protocol proposed in Aleph keeps all of the good properties of HoneyBadger BFT and improves upon it in two important aspects, e.g., tightening the complexity bounds on latency from logarithmic to constant and eliminating the need for a trusted dealer. Aleph is designed for an asynchronous setting, however, it matches the optimistic-case performance of three-round validation time of synchronous protocols [4]. Also, the trustless ABFT randomness beacon generates common, unpredictable randomness.

In more details, a robust atomic broadcast protocol should meet several requirements, such as total order, agreement, and censorship resilience. The properties of total order and agreement are used to ensure that all the honest nodes always produce the same ordering, while the censorship resilience is used to guarantee that no transaction is lost due to censorship, but also guarantee that the system makes progress and does not become stuck as long as new transactions are being received. The Aleph protocol solves Atomic Broadcast over $N >= 3f + 1$ nodes in an asynchronous network, of which f is the number of dishonest nodes, and these nodes have the following properties: 1) Latency – the expected number of asynchronous rounds until a transaction is output by every honest node is $O(N/k)$, where k is the number of honest nodes; 2) Communication complexity – the total communication complexity of the protocol in R rounds is $O(T + R \times N^2 logN)$ per node, where T is the total number of transactions input to honest nodes during R rounds. The second contribution of Aleph is a stand-alone component that allows removing the trusted dealer assumption, which can be considered as a protocol for generating unpredictable common randomness with the ABFT randomness beacon. Such randomness is indispensable in any ABFT protocol for atomic broadcast. Besides, the Aleph protocol is based on a modular structure, which separates entirely the network communication layer from the protocol logic. This is used to maintain a common data structure called a *Communication History DAG*, and the protocol logic is then stated entirely through the combinatorial properties of this structure. Also, Aleph is resistant against the *Fork Bomb*, a spam attack scenario that affects most known DAG-based protocols, through the use of reliable broadcast to disseminate information among nodes.

4) PnyxDB: PnyxDB is a leaderless democratic BFT replicated datastore that exhibits both scalability and low latency, proposed by Bonniot et al. in 2020 [104]. From a high-level perspective, it hinges on a conditional endorsement that tracks conflicts between transactions. PnyxDB also supports application-level voting, e.g., individual ones can endorse or reject a transaction according to application-defined policies without compromising the consistency. Within PnyxDB, clients may perceive conflicting views of the datastore for a short period of time, but they will achieve eventually consistency on different views. With the help of conditional endorsements within quorums, PnyxDB can flag and handle conflicts by

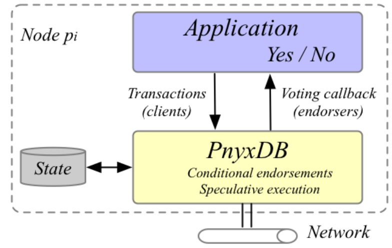


Fig. 7. Overview of PnyxDB. The application submits transactions to be executed on shared state and polls the appplication back for transaction approval before creating conditional endorsements [104].

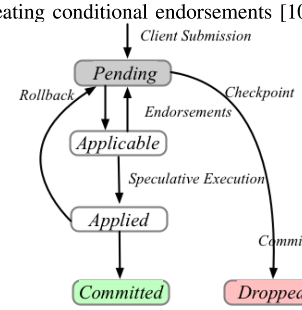


Fig. 8. Transaction state diagram, as viewed by a node. From the Pending state, a transaction evolves either to Dropped or Committed given received messages. Dropped and Committed are eventually consistent across all nodes. However, Pending, Applicable, and Applied are intermediate states local to each node which might not be consistent [104].

allowing each node to specify a set of transactions that must not be committed for the endorsement to be valid. PnyxDB consists of two key components: leaderless quorums for scalability and conditional endorsement for eventual consistency. Fig. 8 shows an overview of PnyxDB, where the application submits transactions to be executed on a shared state and polls the application back for transaction approval before creating conditional endorsement.

In more details, PnyxDB does not rely on any coordinator, rotating or elected, which removes a recurring performance bottleneck. This trades off weaker consistency guarantees for high scalability. Transactions only need to be endorsed by a Byzantine quorum of endorsers (e.g., more than $(n + f)/2$) to be permanently committed to the system's state. The eventual consistency of PnyxDB is based on Conflict-Free Replicated DataTypes (CRDTs) [105] [106] and leverages the fact that many operations in distributed datastores either commute or are independent. This ensures that the transactions can be executed out of order on different nodes without breaking local consistency [107], while allowing every correct node to eventually coverage to the same global datastore state. However, the leaderless quorums may lead to some deadlocks in case of conflicts, e.g., when modifying the same key with conflicting operations. This can be overcome by using the conditional endorsements mechanism. When an endorser broadcasts an endorsement, it publishes a list of transactions that must not be committed for the endorsement to be valid. These conflicting transactions are the conditions of the endorsement. All correct nodes use the same heuristics to decide which one to promote over the other, ensuring a consistent conflict resolution. With cooperative work of leaderless quorums and conditional endorsements, valid transactions can successfully proceed through their light cycle. Fig. 8 shows a transaction state diagram in PnyxDB.

5) Other Leaderless Consensuses: Leaderless Byzantine Paxos is a brief announcement proposed by Lamport in 2011 [95]. It presents a simple method for implementing a leaderless Byzantine agreement algorithm, by replacing the leaders in an ordinary Byzantine Paxos algorithm with a virtual leader that is implemented using a synchronous Byzantine agreement algorithm. Each server decides what message the leader should send next and proposes it as the leader's next message. The servers then execute a synchronous Byzantine algorithm to try to agree on the vector of proposed messages, a vector containing one proposal for each server. Each server uses a deterministic procedure to choose the message sent by the virtual leader, and it acts as if it had received this message. A malicious virtual leader can prevent the progress, but does not cause inconsistency because a Byzantine Paxos algorithm can tolerate a malicious leader. In general, leaderless Paxos adds to a Byzantine Paxos algorithm the cost on the leader agreement algorithm. The time required by a leader agreement algorithm that tolerates F faulty servers is $F + 1$ message delays, which replaces the one message delay of a leader simply sending a message.

Leaderless BFT (LBFT) aims to design a protocol in partial synchronous network model, provided by Niu and Feng in 2020 [93]. From a high-level perspective, all proposers in LBFT can complete proposing blocks once they find some block proofs, and *some* of these blocks will eventually be committed with no explicit leader. The LBFT protocol contains three key components: block proposing, voting, and committing rules. The proposers in LBFT complete to generate new blocks to extend some certified blocks chosen by the proposing rule, in which a proposer is allowed to generate a new block once it finds an associated block proof. The voters typically broadcast votes for the valid block if it satisfies the specified voting rule. When nodes collect more than $2f + 1$ votes for a block, this block will be certified, and nodes will check whether there exist new committed blocks by following the committing rule. For these three rules, interested readers can refer to Section III of [93]. Currently, the LBFT is still on the proof-of-concept stage, and no implementation is proposed to evaluate its performance.

Leaderless consensus aims to provide a general framework for consensus algorithms, proposed by Antoniadis et al. in 2021 [108]. It first gives a precise definition of leaderless algorithms, and then presents three indulgent leaderless consensus algorithms: for shared memory, for message passing with omission failures, and for message passing with Byzantine failures (with and without authentication). In general, this work is much conceptual, which aims to instruct how to design a leaderless consensus. The first proposed leaderless consensus algorithm, called Archipelago, works in shared memory and builds upon a new variant of classic adopt-commit object [109] that returns maximum values to help different processes converge towards the same output. The second algorithm is a generalization of Archipelago in a message-passing system with omission failures. And the third algorithm, called BFT-Archipelago, is a generalization of Arhipelogo for Byzantine failures. BFT-Archipelago shares the same asymptotic communication complexity as a classic BFT consensus algorithm and can execute optimistically a fast path to terminate in two message exchanges under good conditions. Regarding the complexity, BFT-Archipelago can terminate deterministically by exchanging and storing at most $O(n^4)$ messages and bits (each message is of length $O(1)$ bits), and can terminate within $O(n)$ rounds and $O(n^4)$ calculations and signature checks.

C. *Randomized BFT Protocols*

In general, randomization can help to bypass the FLP impossibility by guaranteeing probabilistic properties instead of deterministic ones [27]. Randomization itself cannot be directly used to resolve consensus problems, it must cooperate with other techniques. For example, Robin [110] presents a solution to achieve Byzantine consensus in asynchronous cases by incorporating cryptographic primitives (i.e., digital signatures) and a trusted leader with randomization. On average, it can terminate in a constant expected time. When we discuss the randomized consensus protocols, they typically cannot achieve a deterministic termination. Instead, we consider if they can achieve a consensus "with a high probability". For example, Ben-Or's consensus protocol [111] was the first one to ensure consensus and guarantee termination with a high probability in crash failure model. For the Byzantine model, it is more complex, and most existing randomized Byzantine consensus protocols target asynchronous settings. Thus, some protocols presented in this section also can be categorized into asynchronous BFT consensus.

Different from deterministic protocols, randomized protocols have the ability to achieve the consensus even under a fully asynchronous setting to ensure liveness and responsiveness with a high probability, without any extra assumptions on network synchrony [112]. This also removes complicated timeout schemes over the unstable global communication network (like the Internet). Thus, the randomized algorithms are typically used to bypass the FLP impossibility under asynchronous Byzantine settings. The key idea of a randomized protocol is that it makes progress as if every replica is the leader and retroactively decides on a leader, and this makes the adversary have only a small probability of guessing which nodes to corrupt [113]. Also, most existing randomized protocols have some drawbacks, e.g., a quadratic communication complexity even under good network conditions. This section discusses the randomized BFT consensus protocols.

1) HoneyBadgerBFT: HoneyBadgerBFT is the first practical *asynchronous* BFT protocol that guarantees liveness without making any timing assumption, proposed by Miller et al. in 2016 [66]. By the well-known FLP impossibility, it is impossible to achieve deterministic asynchronous protocols for atomic broadcast and many other tasks. To get a deterministic protocol, it must therefore make some stronger timing assumptions, e.g., the assumptions in partial synchrony. With this regard, almost all modern prior BFT protocols rely on timing assumptions to guarantee liveness. However, in reality, e.g., the Internet, there are no such assumptions. Randomness, e.g., via cryptographic primitives, provides an alternative to achieve a consensus under a purely asynchronous environment. Roughly speaking, HoneyBadgerBFT utilizes a novel atomic broadcast protocol to achieve an optimal asymptotic efficiency, with $O(N)$ communication complexity (where N is the number of total replicas). With HoneyBadgerBFT, it uses a threshold public-key encryption to prevent the targeted censorship attacks, and uses a sub-optional instantiation of Asynchronous Common Subset (ACS) sub-component. With the help of ACS, HoneyBadgerBFT combines efficient reliable broadcast using erasure codes [114] and a reduction from ACS to reliable broadcast from multi-party computation [115]. HoneyBadgerBFT's design is optimized for a cryptocurrency-like deployment scenario where network bandwidth is a scarce resource, but computation is relatively ample. Besides, HoneyBadgerBFT assumes that, in an asynchronous network, messages are eventually delivered but no other timing assumption is made.

In more details, HoneyBadgerBFT consists of two main components: threshold encryption and ACS. In every round, each node chooses a set of transactions as its proposal and encrypts it through a

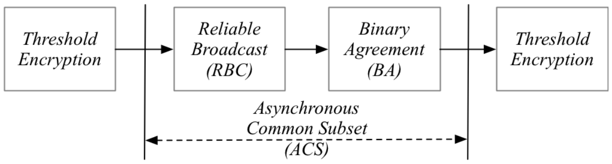


Fig. 9. HoneyBadger BFT [116].

threshold encryption scheme, then submits the ciphertext as input to the ACS module, and a common subset of these ciphertexts will be output. At the end of the round, the subset will be decrypted still by a threshold encryption scheme to get final consensus results [116]. Fig. 9 shows an abstract architecture of HoneyBadgerBFT. The threshold encryption scheme TPKE used in HoneyBadgerBFT is from Baek and Zheng [117]. A (t, n)-TPKE scheme can guarantee that a ciphertext can be decrypted if and only if at least t honest users cooperate. If the number of cooperating users is less than t, no information about the plaintext is leaked. This property is important to applications where a party does not fully trust other individuals, but possibly trusts a group of individuals [118]. By using a TPKE scheme, HoneyBadgerBFT ensures that an adversary controlling less than t faulty replicas cannot get a plaintext without the help of honest nodes. As TPKE is secure under an adaptive chosen ciphertext attack, it helps HoneyBadgerBFT to realize censorship resilience property which prevents the adversary from delaying honest client requests on purpose. ACS serves as the main module to reach consensus among replicas, which further consists of the Reliable Broadcast (RB) module and Binary Agreement (BA) module. In HoneyBadgerBFT, the RBC module from the work [119] with erasure codes [114] is used to transmit the proposal for each node to all other nodes, while the BA module from Mostefaoui [120] is used to decide on a bit vector indicating which proposals should be output as the consensus result.

Besides, the authors implemented the HoneyBadgerBFT protocol, and experimental results show that the system can achieve throughput of tens of thousands of transactions per second, also, it can scale over a hundred nodes over a WAN. Also, the authors conduce experiments over Tor, without needing to tune any parameters. With the guaranteed both safety and liveness, HoneyBadgerBFT has abundant application scenarios, e.g., banks and financial institutions.

There is no timing restriction on message passing. However, HoneyBadgerBFT assumes a traditional static BFT protocol which means that it could not support a consensus node join or leave the consensus group without reconfiguring the whole system. And malicious or inactive nodes still exist in the system, and the current protocol lacks the function to detect these malfunctioning nodes to exclude them from the system. The malfunctioning nodes may hamper the long-term benign working of the system.

2) Algorand: Algorand is a cryptocurrency-based blockchain network that confirms transactions within latency on the order of minutes while scaling to many users, proposed by Gilad et al. in 2017 [121]. It keeps strong consistency that users will never have divergent views on confirmed transactions. This is enforced by a Byzantine Agreement (BA) protocol to reach a consensus among users on the next set of transactions. This also allows Algorand to reach a consensus on a new block with low latency, without the possibility of forks. To achieve scalability, Algorand utilizes verifiable random functions which allow users to privately check whether they are selected to participate in the BA to agree on the next set of transactions, with proof to show they were selected. Users in BA protocol do not require to keep any private state except for their private keys, enabling Algorand to replace participants immediately after they send a message. To applying Algorand for public blockchains, it needs to overcome several challenges, e.g., avoiding Sybil attacks, achieving scalability, and resilience to denial-of-services attacks. To address these challenges, Algorand adopts several techniques, namely, *weighted users*, *consensus by committee*, *cryptographic sortition*, and *participant replacement*. To prevent Sybil attacks, Algorand assigns a weight to each user. $BA\star$ is designed to guarantee consensus as long as a weighted fraction (e.g., a constant greater than $2/3$) of the users are honest. To achieve scalability, $BA\star$ selects a committee, a small set of representatives randomly from a total set of users, to run each step of protocol. And this selection typically is based on the users' weights. All other users observe the protocol messages, which allows them to learn the agreed-upon block. To prevent an adversary from targeting committee members, e.g., DoS attacks, $BA\star$ selects committee members in a private and non-interactive way. Every user itself can independently determine if they are chosen to be on the committee, by computing a VRF function on their private key and public information from the blockchain. Due to the non-interactive nature during member selection, an adversary does not know which user to target until that users start participating in $BA\star$. To prevent an adversary from targeting a committee members, once that member sends a message, $BA\star$ mitigates this attack by requiring committee members to speak just once.

More specially, Algorand grows the blockchain in asynchronous rounds, which is similar to Bitcoin. Each client running Algorand can be considered as an Algorand user, and the communication between users is via a gossip protocol. For example, users can submit new transactions via a gossip protocol, and the other users collect a block of pending transactions that they hear about. In the gossip protocol, each user selects a small fraction set of peers to gossip messages to, and a good gossip protocol can avoid a forwarding loop. Algorand uses $BA\star$ to reach a consensus on one of these pending blocks. In general, $BA\star$ executes in steps, communicates over the same gossip protocol, and produces a new agreed-upon block. $BA\star$ can produce two kinds of consensus: *final* consensus and *tentative* consensus. Once a user reaches the final consensus, any other user that reaches the final or tentative consensus in the same round must agree on the same block value. However, a tentative consensus means that other users may have reached a tentative consensus on a different block (as long as no user reached a final consensus). A user will confirm a transaction from a tentative block only if and when a successor block reaches the final consensus. To participate in consensus, all Algorand users execute cryptographic sortition to determine if they are selected to process a block in a given round. This sortition process ensures that a small fraction of users are selected at random, weighted by their account balance, and provides each selected user with a priority. Meanwhile, the sortition can let Algorand be scalable. Interested readers can refer to the work [121], and some proof works on Algorand can refer [122] [123].

3) ACE: A leader-based view (LBV) abstraction is presented in the work of ACE which provides a general framework for the software design of fault-tolerant SMR systems, proposed by Spiegelman and Rinberg in 2019 [124]. Essentially, ACE provides a simple generic framework for asynchronous boosting, which converts consensus algorithms designed according to the leader-based view-by-view paradigm in a partial synchrony model into randomized fully asynchronous SMR solutions. ACE is a model agnostic, in which it has no model assumptions, and thus can be applied to any leader-based protocol in the Byzantine or crashed failure model. ACE provides a formal characterization of the leader-based view-by-view protocols by defining an LBV abstraction, which encapsulates the core properties of a single view and provides an API that allows decoupling of the leader-based phase from the view-change phase. The view-change phase can be triggered by timers, e.g., parties start a timer in each view, and if the timer expires before the leader drives progress, parties move to the view-change phase. Also, ACE provides a *wave* mechanism to control the trigger of view-change phas. The wave mechanism uses an API provided by the LBV abstraction to generically rearrange views in view-by-view protocols. It composes several LBV abstractions which allows progress at network speed during a period of asynchrony. If the leader of a view is correct and timers never expire, eventually a decision will be made in this view.

4) Validated Asynchronous Byzantine Agreement: Another theoretical work on the optimization of communication complexity is proposed by Abraham et al. in 2019 [125]. It targets to a Validated Asynchronous Byzantine Agreement (VABA) in an authenticated setting. VABA is the key building block in constructing atomic broadcast and fault-tolerant SMR in an asynchronous setting. The proposed VABA protocol has optimal resilence of $f < n/3$ Byzantine failures and asymptotically optimal expected $O(1)$ running time to reach an agreement. Honest parties in protocol send only an expected $O(n^2)$ messages where each message contains a value and a constant number of signatures, whose total expected communication is $O(n^2)$ words. While the best previous result of Cachin et al. from 2001 [126] solves VABA with optimal resilience and $O(1)$ expected time but with $O(n^3)$ expected word communication. Thus, the proposed VABA protocol improves the expected word communication from $O(n^3)$ to asymptotically optimal $O(n^2)$. More specially, the proposed VABA is secure against an adaptive adversary that controls up to $f < n/3$ parities. The overall design is based on a modular implementation, consisting of three sub-protocols: a simple two-round broadcast primitive (called *Provable-Broadcast*), a simple *Leader-Election* protocol, and another primitive (called *Proposal-Promotion*). The Proposal-Promotion is further built on top of sequential instances of Proposal-Broadcast. Interested readers can refer to Section 3 of [125].

5) Gosig: Gosig is a BFT-based blockchain protocol that jointly optimizes the consensus and gossip protocols, proposed by Li et al. in 2020 [127]. Gosig guarantees safety even in asynchronous networks fully controlled by adversaries, by combining secret leader selection with multi-round votings. It adopts transmission pipelining to fully utilize the network bandwidth, and uses aggregated signature gossip to reduce the number of message. These optimizations help Gosig to achieve unprecedented single-chain performance. In general, most existing BFT-based blockchains face several significant challenges: the *targeted-single-point-failure*, in which it is easy for the adversaries to launch DDoS attacks targeting an honest node and partition it from others (e.g., under adaptive attacks), the *limited bandwidth and the long latency*, in which the burdens on the large block and the broadcast scheme may become the source of bottleneck, and the *limitation by the slowest nodes*, in which both the communication and computation overheads on a single node (e.g., the leader) limit the protocol scalability. Even some threshold signatures schemes, like SBFT or HotStuff, help to reduce the total communication overheads (e.g., reducing the total number of messages), however, the nodes need to wait until some collectors receive signature shares from all nodes, in which case the performance is limited by their single node capability. Gosig is a fast scalable and fully decentralized BFT system

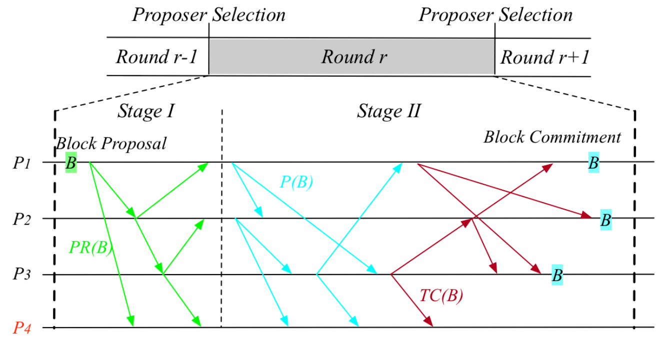


Fig. 10. Overview of a Gosig round without pipelining [127].

that solves the above three challenges by integrating the protocol with the underlying gossip network. Essentially, Gosig operates in rounds, and each round contains two stages: the block proposing stage and the signature collection stage. In the block proposing stage, Gosig first randomly selects several proposers, and then each proposer packs transactions into a new block and broadcasts the block to all other nodes. In the signature collection stage, each node chooses one block received to vote, signs its decision, and keeps relaying the aggregated signatures signed by itself and received from the others. Gosig uses a gossip network to propagate all messages, and the consensus protocol ensures that messages in the same stage can be effectively aggregated during gossiping.

In more details, Gosig adopts some key techniques, e.g., electing a random proposer for every new block, pipelining all possible processes, and aggregating all signatures on the fly using gossip-based broadcast. Gosig uses a secure and cheap proposer replacement to change the proposer for every block. By using a verifiable random function (VRF), it makes the selection randomly and unpredictably. Also, Gosig eliminates the expensive view-change process in standard PBFT, by including a small proof in every proposal to prove a new proposer's validity. Since the protocol proceeds in synchronous rounds, at most one block can be committed in a round. Gosig uses an aggregated signature gossip protocol to optimize the signature collection, which combines multiple received signatures into one aggregated signature. This makes the signature message size be up to two orders of magnitude smaller and significantly reduces the total data transfer during the signature collection stage. When applying the aggregated signature gossip, the vote exchanging is latency-bound, and a significant portion of the network bandwidth is under-utilized. So, Gosig pipelines at both the gossip layer and the BFT voting layer to maximize network utilization.

Gosig maintains a blockchain, and each node relays transactions to others and packs them into blocks in a specific order later. It can tolerate up to f faulty replicas by using $3f+1$ replicas, and safety is guaranteed in an asynchronous network with liveness guaranteed in a partially synchronous network. Gosig proceeds in rounds and each round consists of a proposer selection step and two subsequent stages with a fixed length. Fig. 10 shows an overview of a typical round. At the start of each round, some nodes secretly realize they are *potential proposers* of this round with a cryptographic sortition algorithm, and an adversary cannot target the proposers better than randomly guessing. At stage I, each selected potential proposer packs non-conflicting uncommitted transactions into a block proposal, disseminates it with gossip, and then just like a normal node afterward. At stage II, it aims to reach an agreement on some block proposal by votes exchanging. A node voting a block by adding its digital signature of the block to a prepare message ("*P message*") and gossip it. Upon receiving at least $2f+1$ signatures from P messages for a block, the node tentatively commits on this block, and starts sending tentatively-commit messages ("*TC message*") for it. Once a node receives $2f+1$ TC messages, it finally commits the block to its local blockchain replica. Interested readers can refer to the work [127] for more details.

6) Dumbo: Dumbo is an asynchronous BFT protocol by providing two atomic broadcast protocols to improve the running time asymptotically and practically, proposed by Gao et al. in 2020 [128]. It provides two atomic broadcast protocols, called Dumbo1 and Dumbo2. Both protocols are based on the work of HoneyBadger BFT (as the first practical asynchronous atomic broadcast protocol), and make some improvements on HoneyBadger BFT. Roughly speaking, the asynchronous common subset protocol (ACS) of Dumbo1 only runs a small k (independent of n) instances of asynchronous binary agreement protocol (ABA), while that of Dumbo2 further reduces it to a constant. At the core are two observations: 1) reducing the number of ABA instances significantly improves efficiency; and 2) using multi-valued validated Byzantine agreement (MVBA) which is considered sub-optimal for ACS. The design of these asynchronous protocols is based on two practical requirements: actual network environment and responsiveness. In general, asynchronous protocols may be favorable in a practical environment due to their efficiency, particularly a property called *responsiveness*. A synchronous BFT protocol is typically parameterized by the assumed network latency, which is normally chosen to be large so that the actual network latency is indeed smaller thus the synchrony assumption can be ensured. This causes the efficiency of synchronous BFT protocols depends on the assumed network latency. While the responsiveness should relate only to the actual network latency. Also, asynchronous protocols simplify the engineering efforts, e.g., no timeout mechanism is needed. And the ACS protocol is the core component to achieve an asynchronous agreement under the Byzantine environment. Thus, it is important to carefully design the ACS protocol to improve both the asymptotic and practical efficiency.

More specifically, two ACS protocols are provided to meet these requirements. Based on the FLP impossibility, an ABA protocol must be a randomized version. However, the randomized ABA instances do not really execute in a fully concurrent fashion, as 1) not all instances start at the same time, e.g., some of the instances may start later as inputs have not been delivered, and 2) normal nodes also have an efficiency degradation facing large scale concurrent execution. The Dumbo1-ACS protocol aims to solve these issues. Dumbo1-ACS only needs to run k instead of all n ABA instances, and achieves $O(logk)$ running time, where k is a security parameter independent of n. Simply, the first phase of ACS remains unchanged, e.g., every node broadcasts its input through an RBC instance. In reliable broadcast (RBC) phases, it selects a small number k of nodes as the "leaders" such that at least one of them is honest with overwhelming probability. It needs to take care in the case that two honest nodes may receive different values from different selected "leaders", which requires honest nodes to decide which of the k selected nodes to believe. It relies on an added coin-tossing protocol to select k nodes, which is just one sub-routine of ABA protocol. When one honest node determines to output the values corresponding to S_i, it must be the case that the $i-$th ABA instance outputs a bit 1. And the ABA protocol can ensure that 1) all other honest nodes will also output 1, and 2) at least one honest knows the inputs 1 for this ABA instance. The Dumbo2 uses a mechanism of multi-value validated Byzantine agreement (MVBA) to output one of the inputs of n peers as long as the inputs satisfy some pre-defined predicate. This helps Dumbo2 to achieve optimal (constant) running time, e.g., Dumbo2 only needs to run three consecutive instances of ABA. Since ACS outputs a subset of inputs, it requires preparing each peer node with a vector of inputs via RBC type of protocols. Dumbo2 utilizes a provable reliable broadcast (PRBC) which augments RBC and further outputs a succinct proof (even by a malicious node) that at least one honest peer has received the input. Dumbo2 uses a threshold signature on the RBC index to achieve this. Thus, the ABA implemented in MVBA only needs to be repeated three times to achieve a single instance via the corresponding proofs. Interested readers on detailed design for both Dumboes can refer to the work [128].

Another follow-up work, called Bolt-Dumbo Transformer (BDT), was proposed by Lu et al. in 2021 [112]. It presents a generic framework for optimistic asynchronous atomic broadcast. The framework provides an abstraction of the optimistic case protocol, where can be prepared to design a highly efficient fallback mechanism (compared with MVBA), and can be instantiated easily. It reduces the communication complexity by a factor of $O(n)$ and replaces the cumbersome MVBA protocol with a type of binary agreement only. More specifically, BDT is a randomized asynchronous BFT protocol with optimistic deterministic execution. The optimistic case is taken place if the network is benign, and the Bolt will be run. If no progress can be made using Bolt, in the pessimistic case, an asynchronous BFT protocol Dumbo will start to run. And there is a pace synchronization phase, called Transformer, which will execute if enough honest peers do not make progress and will agree on how to restart. Overall, it provides an abstraction on running specifically designed BFT protocols.

7) Dumbo-MVBA: Dumbo-MVBA is an optimization to reduce the communication complexity of MVBA protocol to the order of $O(n^2)$ (where n is the number of participating nodes), proposed by Lu et al. in 2020 [129]. Original multi-valued validated asynchronous Byzantine agreement (MVBA) [126] requires around the $O(ln^2 + \lambda n^2 + n^3)$ communication (where n is the number of parties, l is the input length, and λ is the security parameter). And later, this communication complexity is reduced by removing the term n^3 when the input is small [125]. However, when the input length $l \geq \lambda b$, the communication is dominated by the λn^2 and the problem of $O(n^3)$ communication remains open. Dumbo-MVBA intends to bridge this gap with $O(ln^2 + \lambda n^2)$ communicated bits, which is optimal when $l \geq \lambda n$. It also maintains other benefits including optimal resilience to tolerate up to $n/3$ adaptive Byzantine corruption, optimal expected constant running time, and optimal $O(n^2)$ messages. At the core of Dumbo-MVBA is an asynchronous provable dispersal broadcast (APDB) in which each input can be split and dispersed to every party and later recovered in an efficient way. Based on the proposed APDB and asynchronous binary agreement, the authors design and present a self-bootstrap framework Dumbo-MVBA★ to reduce the communication cost of existing MVBA protocols.

In more details, the original MVBA protocol only outputs a single party's input, which is unnecessary for every party to send its input to all parties. The design of Dumbo-MVBA achieves a reduction from MVBA to ABA by using a dispersal-then-reset methodology to reduce communication. Instead of allowing each party directly send its input to everyone, Dumbo-MVBA lets everyone disperse the coded fragments of its input across the network. After the dispersal phase has been completed, the parties could randomly choose a dispersed value to collectively recover it. And every party can locally check the validity of the recovered value, so they can consistently decide to output the value or to repeat random election of another dispersed value to recover. A typical MVBA protocol would require an asynchronous verifiable information dispersal (AVID) [130] [131] that needs $O(n^2)$ messages to disperse a value. In Dumbo-MVBA, it uses APDB to weaken the agreement of AVID when the sender is corrupted. By doing so, it can realize a dispersal protocol with only $O(n)$ message complexity. By optimizing the methods in [125] (e.g., combining APDB), each fragment sent by a party has only $O(l/n)$ bits, so n dispersals of $l-$bit input incur only $O(ln)$ bits, and the nodes can reconstruct a valid value after an expected constant number of ABA and recoveries.

D. Asynchronous BFT protocols

Even under the FLP impossibility, there still exist some literature works achieving consensus under an asynchronous setting. These works are typically achieved by relaxing/weakening other conditions on models (e.g., fault-tolerant rate) or adding certain external assistance (e.g., cryptographic primitives). This section discusses some BFT protocols with the feature of the asynchronous setting.

1) Proactive Recover BFT: Proactive recover BFT was first proposed to recover Byzantine-faulty replica proactively by Gastro and Liskov in 2000 [132]. Different from partially synchronous PBFT, this BFT protocol targets an *asynchronous* SMR systems, offering both integrity and high availability in the presence of Byzantine faults. Proactive recover BFT provides two main features: 1) it improves security by recovering replicas proactively; 2) it is based on symmetric cryptography (rather than public-key cryptography) for authentication, which allows it to perform well so that it can be used in practice. The recovery mechanism allows tolerating any number of faults over the lifetime of the system provided fewer than $1/3$ of the replicas become faulty within a window of vulnerability that is small (e.g., a few minutes) under some conditions. The recover algorithm provides *detection* of denial-of-service attacks aimed at increasing the window. For example, replicas can time how long recovery takes and alter their administrator if it exceeds some pre-established bound. The recovery algorithm recovers replicas periodically independent of any failure detection mechanism, thus it allows replicas to continue participating in the request processing protocol while it is recovering.

Even under secure cryptographic co-processors to sign messages, an attacker can still launch replay attacks. To prevent this, proactive recovery BFT utilizes the notion of the authentication *freshness*, and replicas reject messages that are not fresh. To prove to a third party that some messages they received are authentic, the algorithm enables the use of symmetric cryptography for authentication of all protocol messages. This eliminates the use of heavy public-key cryptography which also eliminates the major performance bottleneck. To guarantee a recovering replica up to the date, proactive recovery BFT develops an efficient hierarchical state transfer mechanism based on hash chaining and incremental cryptography, which can tolerate Byzantine faults and state modification while state transfers are in progress. Also, the proposed algorithm ensures both safety and *conditional* liveness. As stated in [132], non-faulty clients eventually receive replies to their requests provided that: 1) at most f replicas become faulty within the window of vulnerability, and 2) denial-of-service attacks do not last forever. Besides, it provides a generic program library with a simple interface, and the experimental results on Network File System (NFS) service show that even a small window of vulnerability has little impact on service latency.

One of the follow-up works on proactive recovery was proposed, named *BFT-PR* (BFT with proactive recovery), by Gastro and Liskov in 2002 [15]. BFT-PR mainly extends their previous work presented in 1999 with more detailed descriptions and detailed implementation on NFS systems with clear interface design. Before the BFT-PR scheme, another work based on proactive recovery BFT was presented in 2001 [133], whose focus is on the performance evaluation of BFT-PR in asynchronous systems. An evaluation on a replicated NFS file system using BFT shows that BFS (Byzantine fault-tolerant NFS file system) performs 2% faster to 24% slower than production implementation of the NFS protocol that is not replicated. We recommend interested readers on proactive recovery can refer to the work [15].

2) FaB Paxos: FaB Paxos is a two-step Byzantine fault-tolerant consensus protocol proposed by Martin and Alvisi in 2006 [134] and its preliminary work is published in 2004 [135]. The protocol can reach consensus under an asynchronous setting in a common case, which can be used to build a replicated state machine that requires only three communication steps per request in the common case. As in classic consensus problems, e.g., [28], FaB Paxos considered three classes of agents: *proposers* who propose values, *acceptors* who together are responsible for choosing a single proposed value, and *learners* who must learn the chosen value. Typically, Paxos is a leader-based protocol, and the leader is responsible for communicating and coordinating with other acceptors. In FaB Paxos, the leader is chosen from the proposers instead of acceptors. And FaB Paxos focuses on improving the performance in a common case scenario, but in a Byzantine model. In general, a common case scenario must meet three requirements: 1) there is a unique correct leader, 2) all correct acceptors agree on its identity, and 3) the system is *in a period of synchrony*. Other processes other than the leader may still fail during the normal cases. For the fault-tolerant rate, FaB Paxos requires $5f+1$ acceptors to tolerant f Byzantine fault in the common case, while its Parameterized FaB Paxos (a generalization of FaB Paxos) only requires $3f+2t+1$ acceptors to tolerate f Byzantine failures and is a two-step scheme as long as at most t acceptors fail.

FaB Paxos works under an *authenticated asynchronous fair links* model, whose model still follows the asynchronous model with some restrictions. Specifically, if a message is sent infinitely many times, then it arrives at its destination infinitely many times, and the recipient of a message knows who the sender is. Under this condition, the consensus may take more than two communication steps to terminate, e.g., when all messages sent by the leader in the first round are dropped. And the authenticated asynchronous fair links models literally enforce an end-to-end retransmission scenario. For example, the caller sends its request repeatedly, and the callee sends a single response every time it receives a request. FaB Paxos requires $5f+1$ processes to tolerate f Byzantine fault in two communication steps. Specially, it requires $5f+1$ acceptors, $3f+1$ proposers, and $3f+1$ learners; and each process in FaB Paxos can play one or more of these three roles. When FaB Paxos is in connection with state machine replication, it assumes that an arbitrary number of clients of the state machine can be Byzantine. Interested readers can refer to the work [134].

Before the FaB Paxos protocol, there exist some similar literature works on Paxos, e.g., FastPaxos (in 2003) [136] and Kursawe's work (in 2002) [71], to optimize the performance as a consensus protocol. Note that "FastPaxos" is not the Lamport's "Fast Paxos" [137] in

2006. In FastPaxos, processes can only fail by crashing, and it only requires $2f+1$ acceptors to tolerate the faulty processes. FastPaxos can also achieve consensus in two communication steps on the eventual leader oracle. Kursawe's optimistic asynchronous Byzantine agreement assumes a Byzantine failure model and operates only with $3f+1$. However, his protocol with much stronger assumptions. For example, to achieve consensus in two communication steps, the proposed optimistic protocol requires that channels are timely and *no* process is faulty. A single faulty process can cause the fast optimistic protocol to switch to a traditional pessimistic slower consensus protocol. Interested readers can read these corresponding papers.

3) Shuttle: Shuttle is a Byzantine Chain Replication (BCR) protocol for asynchronous environments, proposed by van Renesse et al. in 2012 [138]. It provides a framework with the help of an external reconfiguration service. By leveraging an external reconfiguration service, the Shuttle does not base on Byzantine consensus (instead based on BCR), does not require a majority based on quorums during normal operations, and the set of replicas is easy to reconfigure. In practice, the configurations of replicated services are usually managed by an external scalable configuration service, and this configuration service is used only in the face of failures. In the absence of failure, replication protocols that do not rely on quorums can be performed efficiently and robustly. The BCR protocols are based on Chain Replication (CR) [139], which is typically used in scalable fault-tolerant systems. Chain replication uses an external configuration service, but cannot tolerate even general crash failures as the protocol depends on accurate failure detection. Compared with protocols based on CR, the protocols based on BCR are easily reconfigurable, do not require accurate failure detection, and are able to tolerate Byzantine failures. The authors also presented a BCR protocol, called Shuttle.

In more details, in Shuttle, each replica maintains a running state in addition to its local history of order proofs. A centralized configuration service called *Olympus* implements an oracle and generates a series of configurations, issuing an initial configuration statement (a statement with the order proofs) for each instance. A configuration statement includes the sequence number of the configuration and an ordered list of replica identifiers. And the service Olympus generates a configuration upon receiving reconfiguration request statements. To prevent DoS attacks, Olympus only accepts reconfiguration results that are sent by replicas in the current configuration or accompanied by proof of misbehavior. Via such kind of chain service, it can achieve a replication service among replicas. For example, suppose a client wants to execute an operation o to obtain a result, this client is required to first obtain the current configuration from Olympus, and then sends o to the head of the chain. The head orders incoming operations by assigning increasing slot numbers to them, and creating shuttles that are sent along the chain. Typically, a shuttle instance consists of two proofs: an order proof for $< s, o >$, and a result proof for the result of o. The authors describe how to verify these two proofs to provide authentication services. Also, the authors provide two simple implementations (from a theoretical perspective) based on Shuttle: one that can tolerate Byzantine failures in their full generalities, and one that tolerates "accidental failures" such as crashes, bit flips, concurrently bugs, etc. This work focuses on the theoretical models without any practical implementations.

4) Multidimensional Approximate Agreement: Multidimensional approximate agreement in Byzantine asynchronous systems is a $\epsilon-$approximate agreement problem in the aspect of multidimensional (e.g., dimension value $m \geq 1$), proposed by Mendes and Herlihy in 2013 [140]. The authors generalize the $\epsilon-$approximate agreement in Byzantine asynchronous system to consider values that lie in $\mathbb{R}^m$, for $m \geq 1$, and present an optimal protocol with regard to fault tolerance. The proposed work is much theoretical compared to some traditional works on Byzantine fault-tolerant protocols. Following the work of FLP impossibility, the combination of asynchrony and failures that it is impossible to distinguish between a faulty process that has halted (e.g., crashed) and a non-faulty process that is simply slow to respond. There exist many attempts to circumvent this impossibility, and one attempt is to use (uni-dimensional) $\epsilon-$approximate agreement [141]. Compared with strong consensus in the Byzantine environment, the uni-dimensional asynchronous $\epsilon-$approximate agreement is possible, however, it is limited in a uni-dimension. Practically, the multidimensional version can be used in many applications, e.g., robot convergence (applying robots in 2 or 3-dimensional space) and distributed voting (by assigning weights to voting options).

The authors provide a generalization on the ϵ-approximate agreement under asynchrony. In a multidimensional ϵ-approximate agreement task, for arbitrary $\epsilon > 0$ and $m \geq 1$, each process starts with an input value in $\mathbb{R}^m$, and all non-faulty processes must choose output values that meet the following conditions: 1) in $\mathbb{R}^m$; 2) all outputs lie within ϵ of one another; and 3) all outputs lie in the convex hull of the inputs of the non-faulty processes. Based on the above generalization, the authors provide an optimal protocol to solve these problems, satisfying the properties of agreement and convexity. Agreement means that, for any non-faulty processes, the Euclidean distance between their outputs (of two non-faulty processes) must $\leq \epsilon$, an error tolerance fixed a *priori*; while convexity means that, for any non-faulty process, its output is in the convex hull of the non-faulty inputs. Interested readers can read the original paper [140] for more details.

5) BVP: Byzantine Vertical Paxos is a framework design to provide a method to produce a high throughput BFT SMR, tailed for a permissioned blockchain environment, proposed by Abraham and Malkhi in 2016 [142]. It briefly discusses the manifestation in several modes: synchronous, asynchronous, and asynchronous with the help of the Trusted Platform Module (TPM). BVP is a protocol to address both elasticity (dynamic reconfiguration) and throughput, which is based on Vertical Paxos (VP) [143]. VP separates a scheme into two modes: a steady-state protocol and a reconfiguration protocol. The steady-state protocol adopts a primary-backup scenario to multiple nodes via Chain Replication (CR) [139] or Two-Phase Commit (2PC), while the reconfiguration protocol is a Paxos-based reconfiguration engine. By using VP, the steady-state can be optimized for high throughput and the Paxos is used only when reconfiguration is necessary. There exist some prior works, e.g., Byzantine Chain Replication (BCR) protocol [138], used in an asynchronous Byzantine setting. However, BCR requires $2f+1$ replicas for its steady-state and $3f+1$ for the reconfiguration management. The proposed BVP can provide safe progress with as few as $f+1$ replicas with a full-fledged reconfiguration mechanism. For example, in a weak synchronous network, BVP requires only $f+1$ replicas and $2f+1$ replicas for reconfiguration. The core of BVP is to "wedge" a replicated state machine and capture its "closing state". It requires a wedging coordinator to access one non-Byzantine replica participating in a consensus decision.

In more details, the reconfiguration scheme changes the steady-state mode by employing a *wedging* scheme. A wedging coordinator is required to obtain validation from a wedge, a subset of the replicas, then obtain the latest state of the wedge stores. In general, the wedging coordinator is implemented by a *separate* Byzantine consensus engine, *reconfiguration engine*, whose reconfiguration consensus decision has two components: *next configuration* and *closing state*. Once a wedging procedure is complete, a reconfiguration consensus decision is reached. The authors provide two skeleton options for *synchronous-reconfiguration* model: a 4-round (4 message delays) solution with $f+1$ replicas and a 3-round solution with $2f+1$ replicas. For the asynchronous model with a TPM, BVP assumes a weak sequential broadcast (WScast) [144] [145], a broadcast scheme, with the following several features, e.g., FIFO-like same order delivery and eventual delivery among correct replicas. For this model, BVP can achieve a 3-round solution with $f+1$ replicas for a steady model incurring a linear message complexity, and its reconfiguration can be done in constant time using $2f+1$ replicas.

6) XFT: XFT is an acronym of cross fault tolerance, which is an approach to building a reliable and secure distributed SMR system, proposed by Liu et al. in 2016 [10]. In asynchronous BFT SMR, the system designers typically assign an extraordinary power to the adversary, which may control both the Byzantine faulty machines and the *entire network* in a coordinated way. For example, the classic BFT adversary can partition *any number* of otherwise correct machines at will. However, based on the observation in deployed systems, this kind of Byzantine faulty model is a much stronger faulty model, which appears irrelevant in many application scenarios, especially for geo-replicated systems and WANs. From a high-level perspective, XFT can be used to build efficient resilient distributed systems that tolerate both non-crash (Byzantine) faults and network faults (asynchrony). According to the authors' description on XFT, "XFT allows building resilient systems that (a) do not use extra resources (replicas) compared with asynchronous CFT; (b) preserve all reliability guarantees of asynchronous CFT (in the absence of Byzantine faults); (c) provide correct service even when Byzantine faults do occur as long as a majority of the replicas are correct and can communicate with each other synchronously (e.g., when a minority of the replicas are Byzantine-faulty or partitioned because of a network fault)".

In more details, the XFT model relaxes the assumption that an adversary can launch coordinated attacks, which is unlikely in geo-scale replicated system deployments, but it still sufficient to shield applications from crash faults, network faults, and non-crash non-malicious faults. By this relaxed assumption, the XFT model can use the same quorum size and the same number of communication steps as the CFT model by assuming that a majority of processes are correct and asynchronous. In XFT model, the total number of faults are bounded by $t_{nc}+t_c+t_p \leq \lfloor\frac{n-1}{2}\rfloor$, where t_{nc} are the number of non-crash faulty or Byzantine processes, t_c is the number of crash-faulty processes, and t_p is the number of *partitioned* processes. Replica p is partitioned if p is *not* in the largest subset of replicas, in which every pair of replicas can communicate among each other within delay Δ. In any other case, the system will be considered to be in *anarchy*. A protocol is an XFT protocol if it satisfies safety in all executions in which the system is never in anarchy. The above is the major description of the XFT model. In general, XFT can be used in use cases in which an adversary cannot easily coordinate enough network partitions and non-crash-faulty machine actions at the same time. There indeed exist several application scenarios, e.g., tolerating "accidental non-crash faults, wide-area networks, and geo-replicated system, blockchains.

XPaxos is a leader-based SMR protocol based on the XFT model with performance similar to CFT-based Raft/Multi-Paxos [55] protocols. XPaxos specifically targets good performance in geo-replicated settings, which are characterized by the network being the bottleneck, with a high link latency and a relatively low, heterogeneous link bandwidth. Following the generic Paxos protocol, XPaxos consists of a common-case protocol, view-change protocol, a faulty detection mechanism. More details of the basic operations on XPaxos can refer the Section 4 of [10]. However, XPaxos provides poor scalability and performance, partially because XPaxos inherits the shortcomings of the leader-based approaches. For example, leader-based protocols typically have imbalanced load distribution, where the leader does more work than other replicas; there exists high latency for requests originating from non-leader nodes due to the requirement of forwarding requests to the leader; and there exist cases that the inability to deliver any commands whenever the current leader is low or Byzantine pending a view-change message.

A follow-up work based on the XFT model, called Elpis, was proposed by Garg et al. in 2019 [146]. Elpis is a multi-leader XFT consensus protocol. By adapting the Generalized Consensus specification, Elpis exploits the commutativity property inherent in the commands ordered by the systems to provide fast decisions in three communication steps from any node, in the common case. Elpis exploits the workload locality that is very common in geo-scale deployments. The key idea of Elpis is to enable ownership at a finer granularity. For example, instead of having a single leader for ordering all commands (regardless of their commutative property), Elpis assigns ownership to individual nodes such that each node mostly proposes only commutative commands with respect to other nodes. Also, Elpis allows for dynamic ownership changes. For more details on XFT and Elpis, interested readers can refer to the work [146].

7) BEAT: BEAT is a set of BFT protocols for the asynchronous environment, consisting of five asynchronous BFT protocols that are designed to meet different goals, proposed by Duan et al. in 2018 [147]. Essentially, it provides an orchestral of different asynchronous BFT protocols for different purposes, e.g., performance metrics and application scenarios. Each protocol in BEAT is based on the design principle of modularity, and features of these protocols can be mixed to achieve some meaningful trade-offs between functionalities and performance for various applications. In general, asynchronous BFT protocols are appealing solutions to adopt in a WAN environment as these protocols are more resilient to the timing and DoS attacks. Asynchronous BFT typically can ensure the liveness of the protocol without depending on any timing assumption even with a strong adversary. However, existing asynchronous BFT protocols are subject to some challenges, e.g., performance (latency and throughput) issues.

In more details, BEAT leverages secure and efficient cryptographic support and more flexible and efficient erasure-coding support, which provides flexible, versatile, and extensible services, and whose protocols can be designed to meet different needs. BEAT includes five BEAT instances (BEAT0 - BEAT4). BEAT0, BEAT1 are general SMR that can support both off-chain and on-chain smart contracts, while BEAT3 and BEAT4 are BFT storage that can support off-chain smart contracts only. Specially, BEAT0 is a baseline protocol, by incorporates a secure and efficient threshold encryption [148], a direct instantiation of threshold coin-flipping [130], and efficient erasure-code support. BEAT1 additionally replaces an erasure-coded broadcast (AVID broadcast) [114] used on HoneyBadgerBFT with a replication-based broadcast [119]. BEAT2 opportunistically moves the encryption part of the threshold encryption to the client, further reducing latency. BEAT3 is a BFT storage system, whose bandwidth consumption is information-theoretically optimal, and BEAT4 further reduces read bandwidth, which is useful when it is common that clients frequently read only a fraction of stored transactions. Interested readers can read the work [147] for more details.

8) Mir-BFT: Mir-BFT is a BFT total order broadcast protocol aimed at maximizing throughput in WAN, via multiple parallel leaders, proposed by Stathakopoulou et al. in 2019 [149]. It can be deployed in decentralized networks, such as permissioned and Proof-of-Stake permissionless blockchain systems. It allows multiple leaders to propose request batches independently, i.e., parallel leaders, in a way that precludes request duplication attacks by malicious/Byzantine clients, by rotating the assignment of a partitioned request hash space to leaders. The approach of parallel leaders is promising to handle scalability issues, either in a coordinated, deterministic fashion [150] [151], or by using randomized protocols [147] [66] [152]. Mir-BFT removes a single-leader bandwidth bottleneck and exposes a computation bottleneck related to authenticating clients even on a WAN, which can boost throughput using a client signature verification sharding optimization. Essentially, Mir-BFT is a generation of the celebrated and scrutinized PBFT protocol, which follows PBFT "safety-wise" with changes needed to accommodate novel features restricted to PBFT liveness. By using parallel leaders, it needs to prevent request duplication, especially in the form of two attacks: 1) the *request censoring attack* by Byzantine leader(s), in which a malicious leader simply drops or delays a client's request/transaction, and 2) the *request duplication attack*, in which Byzantine *clients* submit the exact same request multiple times. Some naive approaches to solve this issue are: 1) clients are requested to sequentially send to one leader at the time, or 2) simply to pay transaction fees for each duplicate. However, these approaches do not work well in practical settings. In general, Mir-BFT adopts a BFT total order broadcast (TOB) protocol that is to combine parallel leaders with robustness to attacks [153].

In more details, Mir-BFT uses $3f+1$ replicas to tolerate up to f faulty replicas, and can work robustly under arbitrarily long (yet finite) periods of asynchrony. Essentially, Mir-BFT allows multiple parallel leaders to propose batches of requests concurrently, in a sense multiplexing several PBFT instances into a single total order, in a robust way. To achieve that, Mir-BFT partitions the request hash space across replicas, preventing request duplication, while rotating this partitioned assignment across protocol configurations/epochs, addressing the request censoring attack. Also, Mir-BFT uses a client signature verification, sharding throughput optimization to offload CPU. The design of Mir-BFT is to avoid "design-from-scratch", which is known to be error-prone for BFT [14] [80]. It follows the well-scrutinized PBFT protocol and follows the "safety-wise" of PBFT. It also makes some improvements based on PBFT protocol. For example, Mir-BFT uses signatures for requests authentication to avoid concerns associated with "faulty client" attacks compared to MAC authentication in PBFT, in which these signatures can prevent any number of colluding nodes from impersonating a client. Mir-BFT processes requests in batches which can improve the throughput of PBFT. Besides, Mir-BFT makes some improvements on protocol round structure, selecting epoch leaders, handling request duplication and request censoring attacks, and buckets and request partitioning. Interested readers can refer to the work [149] for more details.

E. X-assisted BFT Protocols

X-assisted BFT consensus is typically used to either increase robustness or improve scalability/efficiency. Here "X" can be some software primitives (such as crypto-primitives) or hardware components (such as trusted hardware). The key idea of X-assisted BFT consensus is to guarantee the authenticity of communicated messages. For example, some protocols (e.g., SBFT [65]) are assisted with threshold signature schemes to ensure that there are enough replicas that can collaboratively work on the requests, and some protocols (e.g., Steroids [154]) may consider using a trusted execution environment (such as Intel SGX) as trusted hardware to provision the authenticity of messages. Also, both cryptographic primitives-based approaches and trusted hardware-based approach can be work together to improve efficiency. For instance, FastBFT [155] combines TEEs with a lightweight secret-sharing scheme for efficient message aggregation to achieve scalable Byzantine consensus. This section discusses the works on X-assisted BFT consensus protocols.

1) A2M: A2M is an acronym for *Attested Append-Only Memory* to eliminate equivocation proposed by Chun et al. in 2007 [144]. A2M severed as a trusted system facility that is small, easy to implement, and easy to verify formally. It provides a programming abstraction of a trusted log, which leads to protocol designs immune to equivocation. Equivocation is the ability of a faulty replica to lie in different ways to different clients or servers, and it is the common source of Byzantine headaches. The proposed A2M protocol can an add-on component to add on existing Byzantine fault-tolerant replicated state machines (e.g., PBFT, Q/U. HQ), producing A2M-enabled protocols. Typically, for replicated state machines, the target safety guarantee is *linearizability*, which completes client requests appear to have been processed in a single, totally ordered, serial schedule that is consistent with the order in which clients submitted their requests and received their responses. A2M utilizes a small trusted log abstraction as its primitive to achieve linearizability. A key insight behind A2M is that it provides a mechanism (trusted log) that prevents participants from equivocating, thereby improving the fault-tolerance of Byzantine protocols to f out of $2f+1$. With A2M, once an action is recorded in the log, it cannot be overwritten as A2M did not provide the interface of modification.

The overall design of A2M is based on a classic client-server system, where clients request *authenticated* operations and the server responds to the requesting clients. Its network model is based on the partially synchronous model, where there exists a finite upper bound for message delivery. The fault models of A2M has two cases: *faulty application model*, which the node's owner is well-intentioned but unaware the node's software has been compromised by a third party, and *faulty operator model*, where the node's Byzantine behavior is because of a malicious owner instructing it to do so. For different fault models, A2M has two different trusted computing bases. For the first model, the trusted computing base is set up by the service owner; while for the second model, the owners cannot be trusted, but trust a third party to set up the trusted computing base. Using an A2M implementation within the trusted computing base, a protocol can assume that a seemingly correct host can give only a single response to every distinct protocol request. So, informally, an A2M can be considered equipping a host with a set of trusted, undeniable, ordered logs. An A2M log offers methods to *append* values, to *lookup* values within the log or to obtain the *end* of the log, as well as to *truncate* and to *advance* the log suffix stored in memory. There are no methods to replace values that have already been assigned, as A2M uses cryptography to enforce its properties and to attest the log's contents to other machines. By equipped with A2M in its trusted computing base, reliable service can mitigate the effects of Byzantine faults in its untrusted components, by being able to rely on some small fallback information about individual operations or histories of operations that cannot be tampered with.

A different implementation of the A2M scenarios leads to different threat models and degrees of trust in the resulting system. Typically, according to trusted computing base (e.g., H/W, OS, App), there are five levels of A2M implementation scenarios: trusted service, trusted software isolation, trusted VM, trusted VMM, and trusted hardware. For the details on each scenario, interested readers can refer to Section 3 of [144]. Also, the authors present several state machine replication protocols by integrating A2M to improve their fault tolerance by rendering equivocation extinct or evident. For example, *A2M-PBFT-E* is a simple extension of PBFT that uses A2M logs to protect the execution portion of PBFT, which ensures that replicas cannot equivocate about their locally computed results for a particular requested client operation when replying to that, or any other clients. Another example is *A2M-PBFT-EA* to protect against equivocation during an agreement, in which it requires replicas to append to A2M logs all protocol messages before sending them to their peers. Fig. 11 shows a comparison of different scenarios to equip A2M to PBFT. Besides the above two PBFT based protocols, A2M can be adopted to other Byzantine fault-tolerant systems e.g., SUNDR [156], Q/U [58]. The authors also presented an A2M-Storage, which is an A2M-enabled storage system on a single untrusted server shared by multiple clients. The A2M-Storage provides linearizability in contrast to SUNDR's weaker fork consistency and is simpler than SUNDR.

Despite these improvements on existing replication systems, distributed protocol designers may be reluctant to start assuming the availability of the trusted modules as in A2M. One concern is that the abstraction of a trusted log may require more storage space and complexity than researchers are comfortable assuming, especially for an embedded module inside a potentially hostile component. Another concern is that designers may have difficulty appreciating how broadly applicable a trusted module can be to distributed protocols.

2) TrInc: TrInc is an acronym for *Trust Incrementer*, a small trusted component that combats equivocation in large-scale distributed systems, proposed by Levin et al. in 2009 [157]. TrInc is motivated by a common supposition that individual components in the system are completely untrusted. They must require some trusted technologies to ensure trustworthiness and eliminate the equivocations, e.g., A2M using trusted logs. However, the modules of trusted logs require more storage space and are hard to implement and deploy in a large distributed system. The fundamental security goal of TrInc is to remove participants' ability to equivocate. By using a non-decreasing trusted counter and a key, TrInc provides a new primitive: unique, once-in-a-lifetime attestations. With such primitive, TrInc can run a broader range of protocols, which includes not only client-server systems but all peer-to-peer systems. In general, TrInc provides a smaller size and simpler semantics that make it easier to deploy, as it can be implemented on off-the-shelf available trusted hardware. Also, the TrInc's core functional elements are included in a Trusted Platform Module (TPM) [158] found on many modern devices, which lend credence to the idea that such a component could become available. Besides, TrInc makes use of a shared symmetric session key among all participants in protocol instances, which significantly decreases the cryptographic overhead.

One of the common ways to solve equivocation is to use heavy-communication protocols under a threshold number of faulty participants, such as PBFT. TrInc targets to minimize both the communication overhead and the number of non-faulty participants required. With trusted hardware, it is possible to remove the ability of a (malicious) participant to equivocate without requiring communication among other participants. As a trusted primitive, TrInc must be practical for distributed systems. For instance, a trusted component must be *small* so that it is feasible to manufacture and deploy, and it is difficult and costly to make tamper-resistant for large components. The *trinket* is a such trusted piece of hardware in TrInc. The trinket's API depends only on its internal state, so, unlike a typical TPM, the trinket does not need to access the state of the host devices (e.g., computers). All it needs is an untrusted channel over which it can receive input and produce output.

More precisely, a trinket instance enables an attestation by using a counter that monotonically increases with each new attestation. In such a way, once a certain counter value has been bound to a message, it is impossible to bind a different message to that value. While some applications or protocols may require using multiple counters. Theoretically, anything done with multiple counters can be done with a single counter, but multiple counters allow certain performance optimization and simplifications. The users of a trinket may participant in multiple protocols, each requiring its own counter or counters, and a trinket has the ability to allocate new counters. However, each of them must be uniquely identified so that a malicious user cannot create a new counter with the same identity as an old counter. For performance optimization, TrInc allows its attestation to be signed with shared symmetric keys, which can vastly improve the performance over using asymmetric cryptography or even secure hashes. To ensure participants cannot generate arbitrary attestations, following the mechanism in TPM, the symmetric key is stored in trusted memory (e.g., the enclave of TPM), so that users cannot read it directly. Meanwhile, symmetric keys are only shared among trinkets using a secure communication scheme to ensure that these keys will not be exposed to untrusted parties. Besides, from the implementation and evaluation of TrInc, it shows that TrInc can eliminate most of trusted storage needed to implement A2M, significantly reduces communication overhead in PeerReview, and solves an open incentives issue in BitTorrent [159]. Interested readers can refer to the work [157].

3) MinBFT: Both MinBFT and MinZyzzyva are trust-assisted BFT protocols, requiring only $2f + 1$ replicas to tolerate f faulty replicas, proposed by Veronese in 2011 [81]. Both MinBFT and MinZyzzyva are asynchronous algorithms, which are based on PBFT and Zyzzyva, respectively. To make it clear, we focus on the PBFT version, MinBFT, to discuss its technical merits. MinBFT improves efficiency compared with previous ones in terms of three metrics: the number of replicas, trusted service simplicity, and the number of communication steps. The efficiency of MinBFT mainly comes from the use of a simple trusted component. More precisely, 1) MinBFT requires only $2f + 1$ replicas, instead of the usual $3f + 1$, to tolerate f faulty replicas. 2) The trusted services in which assist to reduce the number of replicas are quite simple, making a verified implementation straightforward and even feasible using commercial trusted hardware. 3) In graceful cases, MinBFT and MinZyzzyva run in the minimum number of communication steps for the non-speculative and speculative algorithms, respectively, four and three steps. Besides the above merits, the algorithms based on hardware are much simpler, being closer to crash fault-tolerant replication

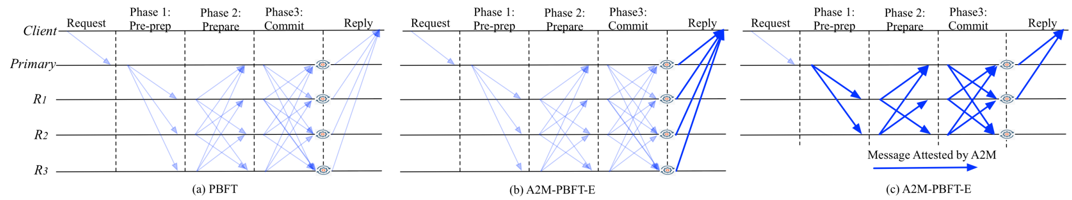


Fig. 11. Three-phase agreement protocol. Thicker lines denote messages that are attested by A2M [144].

algorithms. The authors provided a detailed discussion on these merits with previous ones.

The successful usage of trusted services is based on USIG (Unique Sequential Identifier Generator), which provides an interface with operations only to increment the counter and to verify if other counter values (incremented by other replicas) are correctly authenticated. Essentially, the USIG is a local service that exists in every server, whose duty is to assign messages the value of a counter and signs it. By USIG, the identifiers are unique, monotonic, and sequential for that server. USIG typically has three properties: 1) uniqueness, USIG will never assign the same identifier to two different messages; 2) monotonicity, USIG will never assign an identifier that is lower than a previous one, and 3) sequentiality, USIG will never assign an identifier that is not the successor of the previous one. These three properties are guaranteed even if the server is compromised, so the service has to be implemented in a tamper-proof module or a trusted component. Luckily, the trusted component can be implemented even on commercial off-the-shelf trusted hardware, such as *trusted platform module* [160].

In general, the MinBFT algorithm follows a message exchange pattern similar to PBFT's. The fundamental idea of MinBFT is that the primary uses trusted counters to assign sequence numbers to client requests. Besides the task of assigning a number, the tamper-proof component produces a signed certificate that proves unequivocally that the number is assigned to that message (and not other), and that the counter was incremented, so that the same number cannot be used twice. This is one property of USIG, and each server contains a local trusted/tamper-proof component implementing USIG service. This is used to guarantee that all non-faulty replicas consider that all messages with a certain identifier are the same and, ultimately, agree on the same order for the execution of requests.

MinBFT is a non-speculative $2f+1$ BFT algorithm, which has only two communication steps, not three communication steps like in PBFT or A2M-PBFT-EA. The main role of the primary is to assign a sequence number to each request, and this number is the counter value returned by the USIG service in the unique identifier. These numbers are sequential while the primary does not change, but not when there is a view change. Typically, messages sent by a server are kept in a message log in case they have to be resent. To discard messages from the log, MinBFT uses a garbage collection mechanism based on checkpoints, similar to PBFT's. MinZyzzyva is based on the speculative algorithm Zyzzyva, whose design principle is almost the same as the MinBFT. Besides, the authors present the implementation with some level of isolation for a trusted component used to improve BFT algorithms, and implemented several versions of USIG service with different cryptography mechanisms that are isolated both in a separate virtual machine and trusted hardware. Interested readers on the details of MinBFT and MinZyzzyva can refer to the work [81].

4) CheapBFT: CheapBFT is a resource-efficient BFT system based on a trusted subsystem to prevent equivocation, proposed by Kapitza in 2012 [161]. CheapBFT can tolerate *all but one* of the replicas active in normal case operation become faulty. In general, it runs a composite agreement protocol and exploits passive replication to save resources, and when there are no faults, CheapBFT requires only $f+1$ replicas to actively agree on client requests and execute them during the normal case. When suspected faulty behavior happens, CheapBFT triggers a transition protocol that activates f extra passive replicas, and brings all non-faulty replicas into a consistent state again. From a high-level perspective, the agreement protocol of CheapBFT consists of three sub-protocols: the normal case protocol *CheapTiny*, the transition protocol *CheapSwitch*, and the fall-back protocol *MinBFT*. More specifically, CheapTiny utilizes the concept of passive replication to save resources during normal case operations, which requires only $f+1$ active replicas. In case of suspected or detected faulty behavior of replicas, CheapBFT runs CheapSwitch to bring all non-faulty replicas into a consistent state. When finished the transition of CheapSwitch, the passive replicas will be resumed, and all $2f+1$ replicas become active and execute the MinBFT protocol temporarily to tolerate up to f faults, before eventually switching back to CheapTiny. Essentially, CheapBFT relies on an FPGA-based trusted subsystem, called *CASH*, preventing equivocation.

CASH is an acronym of *Counter Assignment Service in Hardware* to assist CheapBFT for message authentication and verification. To prevent equivocation, each replica in CheapBFT should equip with a trusted CASH subsystem, and it is with initialized a secret key and uniquely identified by a subsystem id, which corresponding to the replicas that hosts the subsystem. To provide trusted counter service, CASH prevents equivocation by issuing message certificates for protocol messages, which consists of the identity *id* of the subsystem, the counter value assigned, and a MAC generated with the secret key. In CASH, it utilizes symmetric-key cryptographic operations for message authentication and verification. The basic version of CASH provides functions for creating (via *createMC*) and verifying (via *checkMC*) message certificate, which targets for single counter cases. To support distinct counter instances and several concurrent protocols, the full version of CASH supports multiple counters, each specified by a different *counter name*. To make CASH practical, it should meet two design goals: minimal trusted computing base and high performance. The code size of CASH must be small to reduce the probability of program errors that could be exploited by attacks, and with each interaction involves authenticated messages, CASH should have a high throughput to meet system requirements. However, the trusted CASH subsystem may fail only by crashing and its key remains secret even at Byzantine replicas.

In more details, CheapBFT is safe in an asynchronous environment. To guarantee liveness, it requires the network and process to be partially synchronous. Fig. 12 shows a CheapBFT architecture with two active replicas and a passive replica (e.g, $f=1$) for normal case operation. For the consensus part, as shown in Fig. 13, each replica follows a total of four phases of communication, which resembles the phases in PBFT. As CheapBFT replicas rely on a trusted subsystem to

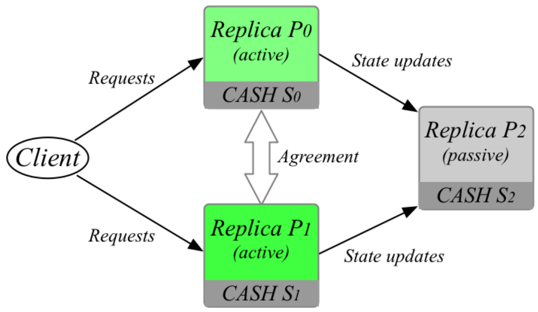


Fig. 12. CheapBFT architecture with two active replicas and a passive replica ($f = 1$) for normal-case operation [161].

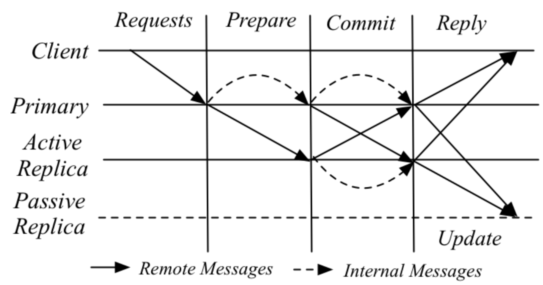


Fig. 13. CheapTiny protocol messages exchanged between a client, two active replicas, and a passive replica ($f = 1$) [161].

prevent equivocation, the CheapTiny protocol does not require a prepare phase. However, with only $f+1$ replicas actively participating in the protocol, CheapTiny is not able to tolerate faults, and it requires MinBFT to provide a more resilient protocol via CheapSwitch. The main task of non-faulty replicas in CheapSwitch is to agree on a CheapTiny *abort history*, and an abort history should be verified to correct even if it has only been provided by a single replica.

5) OMADA: OMADA is a BFT protocol that is able to benefit from additional hardware resources on heterogeneous services, proposed by Eischer and Distler in 2017 [162]. OMADA first parallelizes agreement into multiple groups, and then executes requests handled by different groups in a deterministic order. By varying the number of requests to be ordered between groups as well as the number of groups that a replica participates in between servers, OMADA offers the possibility to individually adjust the resource usage per server. OMADA is based on two practical observations. One observation is that, in traditional BFT protocols, additional replicas typically come at the cost of an increased computational and network overhead, which can further degrade performance. Another is that, traditional BFT protocols assume all replicas run on homogeneous servers, and it is not always possible to operate a BFT system under such conditions, e.g., replicas may run on heterogeneous physical servers. And it is basically impossible for a user to ensure the homogeneity of servers when deploying a BFT system on them. OMADA tries to exploit computing resources that existing protocols are not able to utilize, for example, additional agreement replicas and spare capability on fast servers. To achieve this goal, OMADA parallelizes agreement into multiple heterogeneous groups and varying the agreement workload between them, which allows OMADA to individually adjust the ordering responsibilities of replicas to fit the particular performance capabilities of their servers. For instance, replicas on powerful servers can participate in more than one group that are responsible for ordering a large fraction of replicas, whereas a replica on a less powerful server might only be part of a single group handling a small

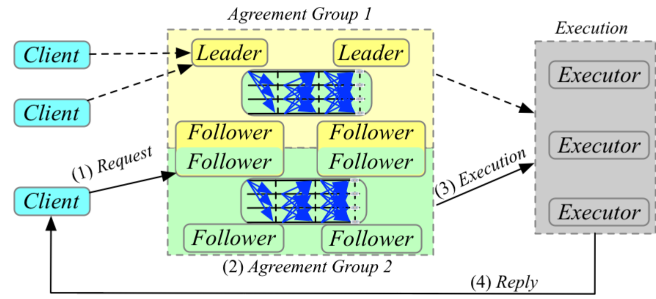


Fig. 14. An overview of the OMADA architecture relying on multiple, possibly overlapping, agreement groups. To invoke an operation at the application, a client (1) sends a request to one of the agreement groups, which then (2) orders the request using PBFT and (3) forwards it to a set of executors. Having processed the request, (4) the executors return their results to the client. [162].

portion of the workload.

In more details, OMADA exploits additional replicas and spare capacities on heterogeneous servers. Fig. 14 shows an overview of the OMADA architecture relying on multiple, possibly overlapping, agreement groups. Even replicas are divided into multiple groups, each group still consists of $3f+1$ agreement replicas. By separating the agreement stage from the execution stage, it only requires $2f+1$ execution replicas. That means, requests do not necessarily need to be processed by the same replicas by which they have been ordered. In OMADA, the roles of replicas can be roughly classified into three categories with different responsibilities: coordinating an agreement group (*leader*), assisting in a group (*follower*), and executing requests (*executor*). One replica of OMADA can participate in more than one agreement group and further assume multiple roles, which allows OMADA to tailor the responsibilities of each replica to the individual performance capability of its server. Despite having multiple agreement groups that operate independently to each other, OMADA is nevertheless able to establish a total order on all client requests. To achieve this, OMADA splits the sequence number space into partitions of equal size and statically maps one partition to each agreement group. The knowledge about the partitions of agreement groups is static and available throughout the system. Thus, it allows a client to randomly select a group at startup which from then on will be responsible for handling all the requests from clients. For more detailed operations of OMADA, interested readers can refer to the work [162].

6) Hybster: Hybster is a hybrid BFT protocol, utilizing a trusted subsystem for message authentication to prevent equivocation, proposed by Behl et al. in 2017 [154]. It requires only $2f + 1$ replicas to tolerate up to f Byzantine faults, with the help of Intel SGX [163]. In general, modern multi-core systems equip with new parallelization schemes, which can help traditional BFT protocol reach unprecedented performance levels. Some latest general-purpose processors can provide a trusted execution environment to protect software components even against malicious behaviors of an untrusted operating system. Hybster is a hybrid SMR protocol that is highly parallelizable and formally specified. Almost all prior SMR systems that are based on a hybrid fault model require sequential processing either of consensus instances that are performed to agree on the order in which replicas must execute commands, or all of the incoming messages. Also, from an abstract perspective, all hybrid systems prevent undetected equivocation by cryptographically binding sensitive outgoing messages to a unique monotonically increasing timestamp by means of the trusted subsystem. However, the features

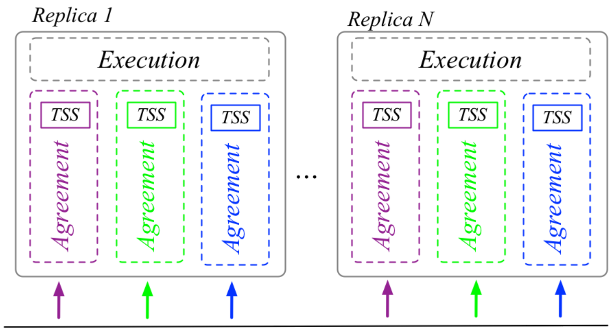


Fig. 15. A parallelizable Hybrid [154].

of parallelism are not well explored. Fig. 15 shows an abstract of a parallelizable Hybster, in which multiple instances can be executed at the same time at some physical replicas. This potentially speeds up the throughput of the system.

In more details, Hybster is designed around a two-phase ordering to prevent equivocation by means of multiple instances of a TrInc-based trusted subsystem realizing using Intel SGX. Though there exist some trusted schemes, such as A2M-PBFT and MinBFT, Hybster is distinct from these schemes in three major ways: relaxed, formal, and parallelizable. For the feature of relaxed, Hybster is based on the observation that it suffices to ensure that only instances must be revealed that are not guaranteed to be propagated by correct replicas, yet have potentially led to the execution of requests. In other words, Hybster allows faulty replicas to conceal their own messages as long as they are not critical to ensure safety, which reduces the complexity of the view-change protocol significantly and prevents histories. For the feature of formal, there is not only an informative description of the basic protocol of Hybster but is also formally specified. For the feature of parallelizable, Hybster has the ability to perform consensus instances in parallel, which benefits from multi-core platforms. Hybster builds on a trusted subsystem abstraction that is akin but not equal to TrInc [157], called TrInX. Hybster is implemented in Java, and with a consensus-oriented parallelization scheme, is optimized to fully exploit multi-core CPUs. It achieves high performance via parallelization, where the performance scales well along with the number of NIC and CPU cores.

7) FastBFT: FastBFT is a fast and scalable BFT protocol with the help of trusted hardware, proposed by Liu et al. in 2018 [155]. Essentially, FastBFT utilizes a message aggregation technique that combines a hardware-based trusted execution environment (TEE) with lightweight secret sharing scheme. From a high-level perspective, FastBFT also combines several other optimizations, e.g., optimistic execution, tree topology, and failure detection, to achieve low latency and high throughput even for large-scale networks. By using message aggregation, it can reduce the message complexity from $O(n^2)$ to $O(n)$, and the message aggregation in FastBFT does not require any public-key operations (e.g., multisignatures), which can further reduce the computation/communication overhead. By a tree topology design in arranging nodes, FastBFT can balance computation and communication load, so that inter-server communication and message aggregation take place along edges of the tree. By the optimistic design, FastBFT only requires a subset of nodes to actively run the protocol. Besides, FastBFT utilizes a simple failure detection mechanism to handle non-primary faults efficiently.

In general, there exist two big categories to improve BFT performance on the fact that replicas rarely fail: speculative and optimistic. The speculative mechanism typically is without running any explicit agreement protocol (e.g., Zyzzyva); while the optimistic mechanism only requires a subset of replicas to run the agreement protocol, and other replicas passively update their states and become actively involved only in case of the agreement protocol fails. The FastBFT protocol adopts an optimistic mechanism. By including a TEE environment, the replicas can remotely verify (e.g., via remote attestation) the behaviors of other replicas. And TEE can only crash but not be Byzantine. FastBFT guarantees safety in asynchronous networks but requires weak synchrony for liveness, and each replica holds a hardware-based TEE that maintains a monotonic counter and a rollback-resistant memory. TEEs can verify one another using remote attestation and establish secure communication channels among replicas. Fig. 16 shows a message communication pattern of FastBFT consensus protocol. Essentially, the consensus protocol of FastBFT consists of four phases: pre-processing, request, prepare and commit. For commit phase, it has two sub-phases that are used to update the state of replicas (a similar procedure for execution phase in traditional BFTs). Besides, the overall FastBFT also includes the failure detection and view-change processes. Interested readers can refer to the work [155].

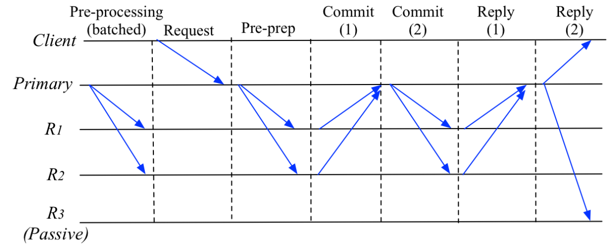


Fig. 16. Message patterns of FastBFT [155].

8) SACZyzzyva: SACZyzzyva is an acronym of Single-Active Counter Zyzzyva to provide resilience to slow replicas and requiring only $3f+1$ replicas, with only one replica needing an active monotonic counter at any given time, proposed by Gunn et al. in 2019 [79]. Speculative BFT protocols, e.g., Zyzzyva, Zyzzyva5, have extremely simple and efficient speculative execution paths when there are no faults or delays. However, they require a trade-off. For example, Zyzzyva requires $3f+1$ replicas to tolerate f faults, but *even a single slow replica* can make Zyzzyva fall back to a more expensive non-speculative operation. Even Zyzzyva5 does not require a non-speculative fallback, but requires $5f+1$ replicas to tolerate f replicas. Realistic communication networks, such as the Internet, are only partially synchronous, a single *slow* but not faulty replica can trigger the non-speculative execution for each protocol run of Zyzzyva, and therefore undermining the efficiency promise of the speculative approach. SACZyzzyva can overcome these drawbacks, which requires only a single replica, the primary, to have an active monotonic counter. Essentially, it can eliminate the need for a non-speculative fallback, and tolerate a subset of replicas being slow while requiring only $3f+1$ replicas. SACZyzzyva utilizes the trusted hardware to some replicas (not all replicas) to assist its process. And this follows a practical setting in real work that only some devices will have the necessary hardware support. Other BFT protocols can also adapt the single active counter approach of SACZyzzyva to reduce the latency while avoiding the need to equip all replicas with hardware-supported monotonic counters.

In more details, SACZyzzyva assumes a weak-synchrony model, and this model enables to analyze liveness during a period of syn-

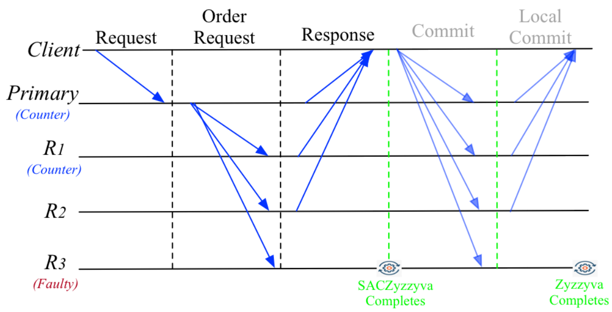


Fig. 17. The communication patterns of Zyzzyva and SACZyzzyva with one faulty replica. Without faults or network delays, Zyzzyva and SACZyzzyva have identical communication patterns, but if any replicas are faulty, Zyzzyva requires two extra rounds of communication [79].

chrony that will eventually occur. Also, SACZyzzyva assumes *some*, but not all, replicas are equipped with a trusted component and, in particular, with a trusted monotonic counter. The goal of SACZyzzyva is to build an efficient SMR protocol that allows replicas to complete a request in the following conditions: 1) in a linear number of messages, and 2) without significant performance reductions in the event of up to f faults. The basic principle of SACZyzzyva is to use trusted monotonic counter in the primary to bind to a sequence of consecutive counter values to incoming requests, ordering requests while avoiding the need for communication between replicas, whether directly or via the client. It does this by signing a tuple consisting of the cryptographic hash of request and a fresh counter value, a single active counter. Thus, SACZyzzyva requires only that $f+1$ replicas have a trusted component, enough that there will always be at least one correct replica that can function as primary.

The communication pattern of SACZyzzyva, as shown in Fig. 17, follows the pattern of the original Zyzzyva. The primary gathers the requests from clients and sends them to all replicas in an order-request message, in which this order-request message is bound to a monotonic counter value to prevent equivocation by the primary. All replicas execute the requests and reply to the client directly if the trusted monotonic counter is sequential to those that the primary has previously sent. If the client receives $2f+1$ replies with matching values and histories, this request can be considered to be complete. Otherwise, the client repeatedly sends the requests directly to the replicas, so that they can detect misbehavior by the primary and so elect a new one. However, this cannot prevent a malicious client. The authors also prove that SACZyzzyva is optimally robust and that trusted components cannot increase fault tolerance unless they are present in greater than two-thirds of replicas.

9) TBFT: TBFT is a TEE-based BFT protocol with inspiration from CFT protocol, which provides a simple and understandable structure, proposed by Zhang et al. in 2021 [164]. Most existing TEE-based BFT protocols are mostly designed via improving traditional BFT protocols and using complex operations to overcome security issues introduced by TEE, whose protocols are difficult to understand and implement. And practically, many protocols eliminated Byzantine failures to crashed failures since the adversary assumption of TEE-based BFT is more similar to CFT rather than traditional BFT. It would be better to design a TEE-based BFT protocol on the basis of CFT protocols to inherit the advantages of CFT, e.g., with a high resilient fault rate. The authors first summarized the key differences between TEE-based BFT and CFT protocols and propose four principles to help bridging the gap between them. Based on these principles, TBFT makes some improvements on both performance and security, including pipeline mechanisms, TEE-assisted secret sharing scheme, and trusted leader election, all of which provides better performance and scalability.

In more details, most protocols assume that TEE may crash but will never provide malicious execution results, which makes TEE-based BFT is more similar to CFT rather than BFT. However, even with the existence of TEE, a Byzantine host can still terminate TEE at any moment, schedule TEE arbitrarily, or drop, reply, delay, reorder, modify the I/O messages of TEE. This simply can be stated that the host in TEE-based BFT may be Byzantine. Thus, it is necessary to bridge the gap between TEE-based BFT and CFT. To bridge the gap, the authors proposed four principles: *one protocol*, *one vote*, *restricted commit*, and *restricted log generation*. For one proposal, the leaders need to call a function, e.g., *create counter* for trusted monotonic counters based TEE, to assign a (c, v) (c is a monotonic counter, v is the current view) for each proposal while other replicas will keep track of the leader's (c, v). Upon calling this function, TEE will increase its local monotonic counter, and hence there will never two different protocols that have the same (c, v). In this way, the leader cannot propose multiple proposals in the same round. For one vote, TEE-based BFT should guarantee that a replica cannot vote for more than one conflicting proposal, and this is achieved by verifying the counter number in (c, v) of the proposal. If matched with its local (c, v), TEE will increase its local monotonic counter, otherwise, reject to vote for it. For restricted commit, the leader should prove that a proposal has been safely replicated to majority replicas by collecting at least $f+1$ votes. For restricted log generation, it is used to handle the limited resource issues within a TEE.

TBFT protocol can tolerate up to f replicas with $2f+1$ replicas, which can guarantee safety in an asynchronous network and guarantee liveness in a partial synchronous network. It follows the above four principles to solve Byzantine behaviors on TEE. In general, TBFT can achieve $O(n)$ message complexity in both normal case and view-change. Also, TBFT utilizes the pipeline scheme to improve throughput, and replace multi-signature with TEE-assisted secret sharing scheme to reduce the computation overhead. Besides, it uses an efficient TEE-based Distributed Verifiable Random Function (DVRF) for verifiable and random leader election. Interested readers can refer to the work [164] for more details.

F. *Unclassified BFT Protocols*

This section provides some unclassified BFT protocols, because either of being less explored research areas or of containing some mixed techniques, as well as some Byzantine faults related literature. After PBFT, many research efforts contributing to the improvement of Byzantine fault-related topics are with a clear focus. Some protocols target on performance and cost issues, e.g., HQ [61], Zyzzyva [59], and AZyzzva [14]. And some other protocols aim to improve robustness, e.g., Ardvark [153] and RBFT [165]. We put all kinds of these protocols in this section, with certain exceptions, e.g., Zyzzyva in classic BFT protocols.

Even though the protocols discussed in this section are categorized as "*unclassified BFT protocols*", we do classify them into different sub-classes by adding a *label* after the protocol name. We provide five labels: *feature*, *architecture*, *theory*, *middleware*, and *blockchain*. The label *feature* means that the discussed protocol typically adds some new features compared with previous protocols, or provides some enhancements on some key features (such as responsiveness,

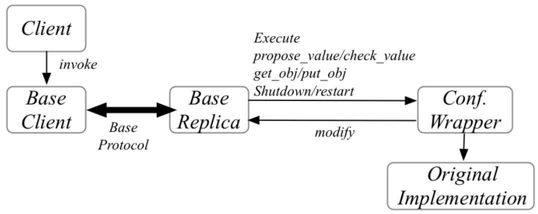


Fig. 18. BASE function calls and upcalls [166].

throughput, and scalability). The label *architecture* means that the discussed protocol typically focuses on the architecture design on existing protocols, and this architecture design typically includes some new communication pattern and new BFT framework. The label *theory* means that the discussed protocol focuses on a theoretical work, e.g., fault tolerance rate under the specified conditions. The label *middleware* typically means that the discussed work is as a middleware to some specific applications and provides the replicated services for these applications. The label *blockchain* means that the discussed work focuses on the construction of blockchain with customized BFT as its consensus protocol. We must note that some protocols have multiple labels. To emphasizes their main contributions, we provide only one label for each discussed protocol.

1) BASE - (architecture): The BASE is an acronym for BFT with Abstract Specification Encapsulation, which was proposed by Rodrigues, Castro, and Liskov in 2001 [166] [167]. The BASE is a replication technique working on data abstraction [168], which uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its ability to mask *software* errors. The abstraction in BASE hides implementation details to enable the reuse of off-the-shelf implementation of important services (e.g., file systems, databases). The original BFT library in [4] [15] provides Byzantine fault tolerance with good performance and strong correctness guarantees, however, it cannot tolerate deterministic software errors, which can cause all replicas to fail concurrently. The BASE scheme improves availability because each replica can be repaired periodically using an abstract view of state stored by correct replicas, and because each replica can run distinct or non-deterministic service implementations, which reduces the probability of common-mode failures.

Fig. 18 shows an abstract BASE *function calls* and *upcalls*. The involved BASE client interacts with BASE replica via BASE protocol, which further interacts with the confirmation wrapper to abstract original implementations. The communication between BASE replica and confirmation wrapper is via well-defined function calls (from BASE replicas to confirmation wrapper) and via upcalls (from confirmation wrapper to BASE replicas).

The key idea of BASE is to utilize the concepts of *abstract specification* and *abstraction function* from work on data abstraction. In general, the BASE offers several advantages. 1) It reuses the existing code base: BASE implements a form of SMR, which allows replication of services that perform arbitrary computations with the feature of determinism. For instance, many implementations produce timestamps based on local clocks, which can cause the states of replicas to diverge. The BASE can resolve this determinism problem by the models of the confirmation wrapper and the abstract state conversions to reuse of existing implementation without modifications. Also, these implementations can be non-deterministic, which reduces the probability of common-mode failures. 2) BASE enables software rejuvenation through proactive recovery: BASE combines proactive recovery with abstraction to counter the problem (as observed in [169]) that there potentially exists a correlation between the length of time software runs and the probability that it fails. Typically, replicas auto-run the proactive recovery periodically to ensure that the service remains available during rejuvenation. For instance, when a replica is recovered, it is rebooted and restarted from a clean state. 3) BASE enables opportunistic N-version programming: N-version programming [170] exploits design diversity to reduce the probability of correlated failures, however, with several issues, e.g., increasing development and maintenance costs, adding unacceptable time delays to the implementation [171]. By abstraction design, BASE enables an opportunistic form of N-version programming, which offers advantages of applying distinct, off-the-shelf implementation of common services.

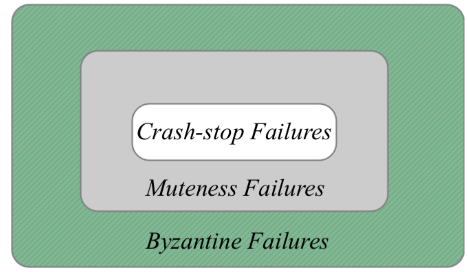


Fig. 19. Inclusion relation between different types of failures [174].

The authors of BASE also built a prototype on the NFS service where each replica can run a different off-the-shelf file system implementation, and a prototype on an object-oriented database where the replicas run the same, non-deterministic implementation. There also exists an extension work with detailed implementation and evaluation published in 2003 [172].

Besides the work on abstraction Byzantine faults, there exist similar literatures to model and encapsulate crash faults to an arbitrary fault. For example, Baldoni, Helary, and Raynal presented a generic methodology to transform a protocol resilient to process crashes into one resilient to arbitrary failure (in the case where processes run the same text and regularly exchange messages) in 2000 [173]. The proposed methodology follows a *modular* approach to encapsulating the detection of arbitrary failures in specific modules. The work can serve as a starting point for designing tools that allow automatic transformation. Interested readers on this transformation process can refer to the work [173].

2) Byzantine Failure Detector - (architecture): An encapsulation-based failure detection for Byzantine failures was proposed by Doudou, Carbinato, and Guerraoui in 2002 [174]. It extends a modular-based *crash* failure detector [175] to a encapsulation-based Byzantine failure detector. The crash failures are the simplest form of failures, and the work [175] has shown the feasibility to detect crash failures, by using *black-box* failure detectors, in the context of a distributed system. However, solving agreement in the Byzantine environment is harder than that in the crash-failure model, since Byzantine detectors need to consider timing assumption as well. The work on the Byzantine failure detector discusses the feasibility of the encapsulation-based approach, and they claim that, in a Byzantine context, it is just impossible to achieve the level of encapsulation of the original crash failure detector model. However, it is possible to get an intermediate approach where algorithms that solve the agreement problem can still benefit from *grey-box* failure detectors by *partially* encapsulating Byzantine failure detection.

The proposed intermediate approach focuses muteness failures. A muteness failure can be viewed as a special case of Byzantine failures, yet is more general than crash failures, as shown in Fig. 19. It encompasses the physical and algorithmic crashes. And muteness failure detectors encapsulate a sufficient level of synchrony to solve a consensus problem under Byzantine assumption, however, the specification of muteness failure detectors is not orthogonal to the algorithms using them. Besides, on the consensus synchrony model, the proposed failure detector implementation is based on a partially synchronous model.

To design a successful Byzantine failure detector, several aspects are required to consider. 1) Specification orthogonality. Due to inherent nature of Byzantine failure, it is impossible to design a failure detector that is independent of the agreement algorithms (such as consensus or atomic broadcast). Instead, the proposed work looks for an intermediate approach to detect *muteness failures*, which can still capture the tricky part of Byzantine failures and restrict the dependency between the algorithms and the failure detector. 2) Implementation generality. A crash failure detector can be considered a black-box, of which implementation can change without any impact on the algorithm that uses it. However, the Byzantine failure does not have this feature. Thus, the proposed work switches a *grey-box* approach with a parameterized implementation of a muteness failure detector that is capable to solve consensus problems in a Byzantine environment. 3) Failure encapsulation. In a Byzantine context, some failures cannot simply be encapsulated inside a failure detector (e.g., timing-related issues), which should rely on some additional modules to handle these failures. The proposed work uses the instance of each broadcast and certification protocol to solve these failures. 4) Failure transparency. The failure of transparency is simply impossible in the Byzantine context which requires external resources to assist the process of transparency failure. The proposed work utilized an atomic broadcast algorithm to achieve this, e.g., some failures are masked by the consensus algorithm. Interested readers on the model definitions and detection methodologies can refer to the work [174].

3) Separating Agreement from Execution - (architecture): An architecture on Separating Agreement from Execution for BFT (SAE-BFT) is proposed by Yin et al. in 2003 [176]. Traditionally, an SMR scheme relies on first agreeing on a linearizable order of all requests, and then executing these requests on *all* state machine replicas. And, it is a common practice to tightly couple these two functions together, however, coupling both agreement and execution is not so effective considering confidentiality. The key principle of SAE-BFT is to *separate agreement from execution*, and this separation can clearly yield some fundamental advantages. On the one hand, this separation architecture can reduce replication costs, as this architecture can tolerate faults up to half of the total of replicas that are responsible for *executing* requests. However, the system still requires $3f + 1$ *agreement replicas* to *order* requests using $f-$resilient Byzantine agreement. This means, it only requires a simple majority of correct *execution replicas* to process the ordered requests. The choice of doing this separation is based on the fact that execution replicas are likely to be much more expensive than agreement replicas, from both the hardware and software perspectives. Even the *n-version* programming helps to some extent, however, it will increase the cost of each different service. On the other hand, this separation architecture leads to a practical and general *privacy firewall* architecture to protect confidentiality through Byzantine replication. Though traditional replication architectures (e.g., PBFT) can achieve integrity and availability [25], they cannot ensure confidentiality, in which a malicious client is allowed to observe the confidential information leaked by faulty servers.

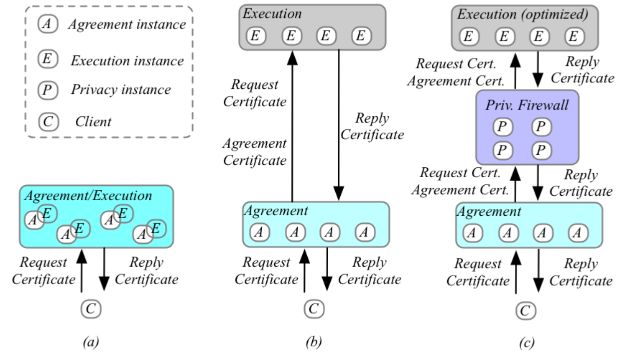


Fig. 20. High level architecture of (a) traditional Byzantine BFT SMR, (b) separate BFT agreement and execution, and (c) optimization-enabled by separation of agreement and execution [176].

Fig. 20 shows an abstract of the high-level architectures of (a) traditional Byzantine fault-tolerant SMR (combining both agreement and execution together), (b) separate Byzantine fault-tolerant agreement and execution, and (c) an optimized version on the separation of agreement and execution [176]. The pre-requisite to separated architectures (showing in both Fig. 20(b) and Fig. 20(c)) relies on some cryptographically verifiable primitives, and the exchanged certificate can be verified by any server. Thus, it is possible to separate the execution replicas from the agreement replicas.

SAE-BFT can function on a distributed asynchronous system with the guarantee of safety only. Due to the FLP impossibility, it is impossible to guarantee liveness under asynchronous settings unless some assumptions are made on communication synchrony and message loss. However, the system can make progress under some relatively *weak bounded fair links*. From the definition of *fair links*, we can literately consider these links are one branch of partially synchronous cases. Interested readers on these assumptions can refer to the work [176]. Besides, the authors also built a prototype system to evaluate the proposed architecture. The system utilizes a threshold signature scheme to guarantee confidentiality. The evaluations on both micro-benchmarks and an NFS server show that the architecture adds modest latency compared with the unreplicated systems, and the evaluated performance is competitive with the existing (of the year 2003) Byzantine fault-tolerant systems.

4) BART - (architecture): BART is an acronym for *Byzantine Altruistic Rational (BAR) Tolerant* for cooperative services proposed by Aiyer et al. in 2005 [177]. Typically, the cooperative services span multiple administrative domains (MADs), and the BAR model in cooperative services should accommodate three classes of nodes. Altruistic [178] nodes follow the suggested protocol exactly. Intuitively, altruistic nodes correspond to *correct nodes* in the fault-tolerant literature. Rational [179] nodes participate in the system to gain some net benefits and can depart from a proposed program in order to increase their net benefits. These nodes are self-interested and seek to maximize their benefits according to a known utility function. And Byzantine nodes can depart arbitrarily from the proposed program whether it benefits them or not. The BART protocol provably provides its non-Byzantine participants with the desired safety and liveness properties. The proposed BART protocol assumes that at most $(n-2)/3$ of the nodes in the system are Byzantine and that every non-Byzantine node is rational, which does not depend on the

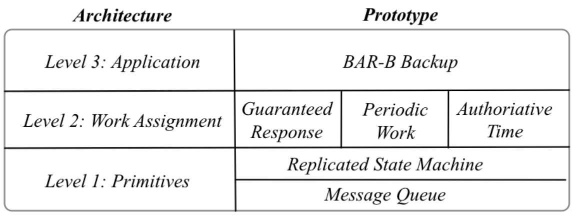


Fig. 21. Three-level architecture of BART service [177].

existence of altruistic nodes in the system.

To achieve a cooperative service, a general three-level architecture for BART services is built, as shown in Fig.21. The bottom level, *basic primitives* level, implements a small set of key abstractions, e.g., state machine replication and terminating reliable broadcast, that simplify implementing and reasoning about BART distributed services. The middle layer, *work assignment* lever, partitions and assigns work to individual nodes, whose assignment is done through a *Guaranteed Response* protocol that generates either a verifiable match between a request and the corresponding response or a verifiable proof that a node failed to respond to a request. The top-level, *application* level, implements the application-specific aspects of BART services, e.g., verifying that responses to requests confirm to application semantics. In general, the lower levels provide reliable communication and authoritative request-response bindings, while the application is responsible for providing a net benefit and defining legal request-response pairs.

The authors also implemented BAR-B, a BART cooperative backup service. The BART protocol assumes that there exists a trusted authority to control which nodes may enter the system, e.g., using cryptographic public keys as their unique identities. For BAR-B and the underlying BART replicated state machine, they have different timing assumptions. For example, BAR-B relies on synchrony to guarantee both its safety and liveness, and data trusted to BAR-B is guaranteed to be retrievable only until the lease associated with it expires. While the underlying BART state machine is safe even in an asynchronous system, though liveness is only guaranteed during periods of synchrony. The BRAT asynchronous replicated state machine (RSM) is based on PBFT with modifications motivated by the BAR model. These modifications are based on four guiding principles (from a high-level perspective): ensure long-time benefit to participants, limit non-determinism, mitigate the effects of residual non-determinism, and enforce predictable communication patterns. Interested readers on these principles and detailed BART protocol can refer to the work [177].

A follow-up work on BRA model, called *BAR Primer*, is proposed by Clement et al. in 2008 [180]. The authors formalize a classic synchronous repeated terminating reliable broadcast (R-TRB) problem as a game, which leads towards a provably BAR-tolerant solution. It focuses on some proof work to show the safety of the proposed BAR-tolerant R-TRB protocol.

Besides the Byzantine fault-tolerant protocol based on the BAR model, there exist some other works on gossip protocols based on the BAR model, e.g., BAR Gossip [181] in 2006. Gossip algorithms were first introduced by Oppen et al. to manage replica consistency in the Xerox Clearinghouse Services [182]. Now it is developed for the use cases in P2P networks. BAR gossip is a P2P streaming media application designed for a BAR model, which describes a method for an altruistic *broadcast* to stream a live event to a pool of clients. It provides a salable mechanism for information dissemination that ensures predictable throughput. Technically, BAR Gossip relies on *verifiable pseudo-random partner selection* to eliminate non-determinism that can be used to game the system while maintaining the robustness and rapid convergence of the traditional gossip. The robustness of BAR Gossip is enhanced by the randomness to randomly select partners. Also, a *fair enough exchange* primitive entices cooperation among selfish nodes on short timescales, avoiding the need for long-time node reputations as well as avoid the free rides. Interested readers on how BAR Gossip achieves consistency can refer to the work [181].

5) HQ - (architecture): HQ is a hybrid Quorum-based Byzantine fault-tolerant state machine replication protocol proposed by Cowling et al. in 2006 [61]. Typically, there exist two approaches to providing replication service under Byzantine faults: the replica-based approach (aka. agreement-based approach), e.g., BFT, that uses communication between replicas to agree on a proposed ordering of requests; and the quorum-based approach, such as Q/U, in which clients contact replicas directly to optimistically execute operations. For replica-based approaches, e.g., BFT, the quadratic cost of inter-replica communication is unnecessary when there is no contention, while for quorum-based approaches, e.g., Q/U, it requires a large number of replicas and performs poorly under contention. HQ employs a lightweight quorum-based protocol when there is no contention, but uses BFT to resolve contention when it arises. And HQ requires only $3f + 1$ replicas to tolerate f faults, which provides optimal resilience to failure nodes. Specifically, when no contention occurs, HQ utilizes a quorum protocol in which reads require one round trip of communication between client and replicas, and writes require two round trips. However, when contentions occur, it uses the BFT state machine replication algorithm to efficiently order the contending operations.

In HQ, both clients and servers can be Byzantine nodes, which may deviate arbitrarily from their specifications, and it is assumed that an asynchronous distributed system with the network is fully connected. The operations of HQ are phase-based, e.g., two phases for a write operation and one phase for a read operation when no failure and no contention occur. Each phase consists of a client issuing an RPC call to all replicas and collecting a quorum of replicas. The HQ protocol requires $3f + 1$ replicas to tolerate f failures, by using the quorums of size $2f + 1$. The authors also made some improvements on the communication scheme among replicas of BFT system, by using TCP instead of UDP, to avoid costly message loss in case of congestion. Also, the use of TCP can avoid exponentially back-off resolution to contention which greatly leads a reduced throughput. The other improvement is the use of MACs instead of authenticators in protocol messages. Also, the improved BFT is allowed to use the scheme of preferred quorums. However, on the batch process to improve the throughput, HQ cannot batch concurrent client requests as they do not have a primary replica funneling all requests to other replicas. As shown in the evaluation of [62], the inability to support the batching scheme increases the cryptographic overhead per request at the bottleneck node, by factors approaching 4 when one fault ($f = 1$) is tolerated, and by higher factors in system that tolerates more faults. Interested readers can refer to these detailed protocols and optimizations in [61].

6) BFT2F - (theory): BFT2F is an extension of PBFT protocol to explore the design space beyond f failures (of $n = 3f + 1$ total replicas, a constant number), proposed by Li and Mazieres in 2007 [183]. The “2F" in BFT2F means it can tolerate up to $2f$

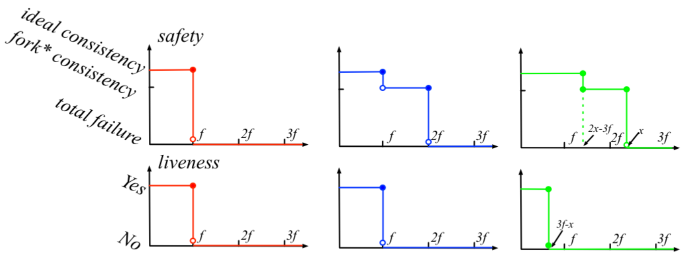


Fig. 22. Comparison of the safety and liveness guarantees of PBFT, BFT2F, and BFTx. As we can see, BFT2F provides extra safety guarantees without compromising liveness, which is strictly better than PBFT [183].

faulty replicas out of $3f+1$ with some assumptions. Typically, most BFT protocols make a strong assumption that some predetermined fraction of server replicas are honest. For example, the highest fraction of failure that an asynchronous BFT system can survive without jeopardizing safety or liveness is f out of $3f+1$ replicas. The reason as stated in the FLP impossibility is that asynchronous communication makes it impossible to differentiate slow replicas from failed ones. Thus, to progress safely with f unresponsive replicas, the majority of the remaining $2f+1$ responsive ones must be honest. The goal of BFT2F is to limit the damage when more than f out of $3f+1$ servers in a replicated SMR fail. To achieve this goal, the authors deviated to a weaker consistency model, called $fork\star$ consistency, based on previous work on fork consistency of PBFT. In general, even without $fork\star$ consistency, BFT2F has the same liveness and safety guarantee as PBFT when no more than f replicas fail. With $fork\star$ consistency, it is possible to bound a system's behavior when between $f+1$ and $2f$ replicas have failed. When $2f+1$ or more replicas fail, it is unfortunately not possible to make any guarantees without simultaneously sacrificing liveness for cases where f or fewer replicas fail.

Fig. 22 shows a comparison of the safety and liveness guarantees of PBFT, BFT2F and BFTx (where $2f < x <= 3f$). When more than f but no more than $2f$ replicas failed, two outcomes are possible. 1) System may cease to make progress, in which BFT2F does not guarantee liveness when more than f replicas are compromised. 2) System may continue to operate and offer $fork\star$ consistency, which is a preferable option compared to arbitrary behaviors. The case of up to $2f$ malicious replicas failed is equal to the case where a *majority* of the replicas may fail. Also, BFT2F can be easily extended to a parameterized version, as shown in term BFTx of Fig. 22. When the number of faulty replicas is greater than f, the results highly depend on the $fork\star$ consistency, which is a weaker consistency model and has a chance to fail to achieve the correct consistency. The $fork\star$ consistency can be achieved in a single-round protocol. The major difference compared with fork consistency is that each request specifies the precise order in which the same client's *previous* request was supposed to have been executed. Doing so may possibly make an honest replica execute an operation out of order, however, at least any future request from the same client will make the attack evident. Interested readers can refer to the work [183] for the details on the $fork\star$ consistency and the BFT2F algorithm.

7) HRDB - (middleware): HRDB is an acronym for Heterogeneous Replicated Database to tolerate Byzantine faults in transaction processing systems using a concurrency control protocol proposed by Vandiver et al. in 2007 [184]. As stated in the paper, the main challenge in designing a replication scheme for a transaction processing system is to ensure that different replicas execute transactions in equivalent serial orders while allowing a high degree of concurrency. A transaction processing system functioning errors may experience different faults, e.g., crash faults or Byzantine faults. In cases of Byzantine faults, examples include concurrency control errors, incorrect query executions, a database table or index corruption, and so on. HRDB is a replication scheme to handle both Byzantine and crash faults in transaction processing systems, without modification to any database replica software. To design such a system, two goals should be met: correctness (e.g., providing a single-copy serialization view of database) and high performance (e.g., performance not worse than a single database). Technically, achieving both goals simultaneously is challenging because databases achieve high performance through concurrency. While concurrency may cause inconsistency, which further breaks the correctness of concurrent systems. HRDB presents a concurrency control protocol, called *commit barrier scheduling (CBS)*, that allows the system to guarantee a correct behavior while achieving its high concurrency. CBS constrains the order in which queries are sent to replicas just enough to prevent conflicting schedules while preserving most of the concurrency in a workload.

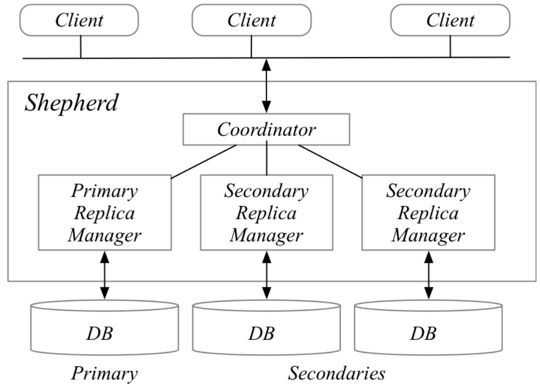


Fig. 23. HRDB system architecture [184].

Fig. 23 shows an abstract of high-level HRDB system architecture. In HRDB, clients do not interact directly with database replicas, instead of communicating with a *shepherd* which acts as a front-end to the replicas and coordinates them. All transactions that run on the replicas are sent via the shepherd, and the shepherd is assumed to be trusted and does not have the Byzantine fault. In HRDB, it requires only $2f+1$ database replicas to tolerate f number of simultaneously faulty replicas since the replicas do not carry out the agreement and simply execute statements sent to them by the shepherd. The shepherd runs a single coordinator and one replicas manager for each back-end replica. To achieve the "ACID" semantics on transaction processing, the shepherd implements a commit barrier scheduling scheme (CBS). The CBS ensures that all non-faulty replicas execute committed transactions in equivalent serial orders, while at the same time preserving much of the concurrency of the individual replicas. In CBS, one replica is designated to be the *primary*, and runs statements of transactions slightly *in advance* of the other *secondary* replicas. The order in this process on the primary determines a serial order, a kind of vector. And the shepherd observes which queries from different transactions are able to execute concurrently without conflicts at the primary, and allows the secondaries to execute these queries concurrently. Technically, the CBS schedule highly depends on the assumption that replicas use a strict two-phase locking scheme to ensure correctness.

In brief, HRDB uses a trusted node to coordinate the replicas. And the coordinator chooses which requests to forward concurrently to maximize the amount of parallelism between concurrent requests. According to the evaluation on practical databases, HRDB provides a good performance, however, it requires trust in the coordinator, which might be problematic if replication is being used to tolerate attacks. Besides, HRDB can ensure the single-copy serializability, in which the group of replicas must together act as a single copy if no more than half of the replicas are faulty. We refer interested readers to read the work [184] for detailed CBS scheduling.

8) Erasure-coded BFT - (feature): An erasure-coded BFT protocol is proposed for block storage by Hendricks in 2007 [185]. Different from replicated state machines-based BFT, in which each request is sent to a server replica and each non-faulty replica sends a response, an erasure-coded BFT protocol relies on the erasure codes. Taking *m-of-n* erasure code as an example, each block is encoded into n fragments such that any m correct fragments can be used to decode that block. To avoid expensive overhead, e.g., on communication, the authors proposed a mechanism to optimize for the common case when faults and concurrency are rare. This optimization minimizes the number of rounds of communications, the number of computations, and the number of servers that must be available at any time. One novelty of this work is that the block is erasure-coded at the server, and this will eliminate the verification of concurrency on clients. Also, the proposed scheme utilizes a homomorphic fingerprinting technique to ensure that a blockchain is encoded correctly, and the homomorphic fingerprinting technique can eliminate the need for the versioned storage and a separate garbage collection protocol.

PASIS is another Byzantine fault-tolerant erasure-coded storage protocol proposed by Goodson et al. in 2004 [186]. It describes a decentralized consistency protocol for survivable storage that exploits local data versioning with each storage node. By exploiting versioning storage nodes, the protocol shifts most work to clients and allows highly optimistic operations, e.g., reads occur in a single round-trip unless clients observe concurrency or write failure. In PASIS, servers do not verify if a block is correctly encoded during a write operation. Instead, clients verify during read operations that a block is correctly encoded. Besides, PASIS requires $4f + 1$ servers to tolerate f faulty servers.

AVID is an asynchronous verifiable information dispersal protocol proposed by Cachin and Tessaro in 2005 [114]. It is also an erasure-coded storage protocol for Byzantine fault tolerance, which requires only $3f + 1$ replicas to tolerate f faulty replicas. In AVID, a client sends each server an erasure-coded fragment, and each server then sends its fragment to all other servers such that each server can verify that the block is encoded correctly. Specially, it combines an asynchronous reliable broadcast protocol with erasure coding to achieve efficient communication. However, essentially, this is still an all-to-all communication, which consumes slightly more bandwidth in the common case than a typical replication protocol.

Compared with replication-based Byzantine fault-tolerant protocols, the erasure-coded Byzantine fault-tolerant scheme is more suitable for storage operations to avoid case conflicts on writes and the corresponding concurrency issues. While the replication-based BFT systems focus on the consistency of the states among replicas to ensure their safety and liveness. Interested readers on erasure-coded BFT schemes can refer to these above-mentioned papers.

9) Prime - (feature): Prime is an acronym for Performance-oriented Replication In Malicious Environments, an SMR protocol to resilient to performance degradation under attacks, proposed by Amir et al. in 2008 [187] and its extended version in 2010 [188]. Practically, faulty processes can significantly degrade performance by causing the system to make progress at an extremely slow rate, and the systems that are vulnerable to performance degradation are of limited practical use in an adversarial environment. Prime is a Byzantine fault-tolerant SMR protocol to mitigate performance attacks and bridge this “practicality gap" for intrusion-tolerant replication systems. In Byzantine environments, faulty processors can exploit the vulnerability on “stable periods" during synchrony to degrade system performance to a level far below what would be achievable with only correct processors. For example, a small number of faulty processors, especially if containing the leader of leader-based BFT protocols, can cause the system to make progress at an extremely slow rate. Prime explores this kind of performance degradation attacks, and called them Byzantine performance failure. This failure is different from the traditional classification of Byzantine failures, such as failures in the *value* domain (e.g., sending incorrect or conflicting messages) or in the *time* domain (e.g., timeout of messages). Processors exhibiting performance failures can send correct messages slowly but without triggering protocol timeouts. Thus, Prime proposes a performance-oriented metric to evaluate Byzantine failures and presents an SMR protocol that performs well under the metric.

Prime focuses on classic leader-based Byzantine SMR protocols, in which they rely on a leader to coordinate the global ordering and are thus vulnerable to performance degradation caused by a slow leader. In general, Prime can guarantee the safety and validity of all executions, including those in which the network is asynchronous and may drop or duplicate messages. Validity means only an update that was proposed by a client may be executed. Since no asynchronous Byzantine agreement protocol can always be both safe and live, Prime guarantees liveness only in execution in which the network eventually meets certain stability conditions (e.g., PRIME-STABILITY defined in [188]). The PRIME-STABILITY puts some constraints on the minimum latency on messages transmission between correct replicas. Under PRIME-STABILITY, PRIME-LIVENESS can be guaranteed. Compared with the traditional liveness in existing leader-based protocols, PRIME-LIVENESS only requires a strong degree of stability only. Also, under PRIME-STABILITY, Prime can provide a stronger performance guarantee, which is called BOUNDED-DELAY. BOUNDED-DELAY states that there exists a time after which the update latency for any update initiated by a stable server is upper-bounded. However, resource exhaustion denial of service attacks may cause PRIME-STABILITY to be violated for the duration of attacks. And, Prime handles the case that malicious leaders can slow down the system without triggering defense mechanisms, and focuses on two scenarios: Pre-prepare delay attack and timeout manipulation attack. In Pre-prepare delay attack, a faulty server delays the ordering of requests from some of the clients, causing a considerable increase in the latency of these requests and a great reduction of the throughput. In the timeout manipulation attack, faulty servers manage to increase the timeout used in PBFT, seriously degrading the performance of the system. Also, these attacks apply to some of the algorithms that derive from PBFT. Prime tolerates these attacks by adding a pre-ordering of three communication steps to BFT. In general, defending against these two performance attacks allows Prime to meet BOUNDED-DELAY.

Prime requires $3f + 1$ servers to tolerate f Byzantine faults, and the Prime protocol includes two main components: Prime ordering protocol and detecting malicious leaders. In Prime ordering protocol, it consists of three phases: preordering phase, global ordering phase, and reconciliation. To detect malicious leaders, three mechanisms are adopted: enforcing up-to-date Pre-preparing messages, Pre-prepare flooding, and suspect-leader protocol. We refer interested readers to read the protocol and detection mechanisms in details [188].

10) Nysiad - (architecture): Nysiad is a system that transforms a scalable distributed system or network protocol tolerating only crash failures into the one that tolerated Byzantine failure, proposed by Ho et al. in 2008 [189]. The key idea of Nysiad is to assign each host a certain number of *guard hosts*, optionally chosen from the available collection of hosts, with assumptions that no more than a configurable number of guards of a host are faulty. Nysiad then enforces that a host either follows the system's protocol and handles all its inputs fairly, or ceases to produce output messages altogether. Nysiad leverages the case that most distributed systems already deal with crash failures, and translates arbitrary failures into crash failures, so that the translated system can tolerate arbitrary failures in the content of crash failures. By doing this translation, it can avoid solving consensus during the normal operation (e.g., getting an agreement among replicas). However, Nysiad does need a consensus only when a host needs to communicate with new peers or when one of its replicas is being replaced. Rather than treating replicas as symmetric (e.g., in most SMR systems), Nysiad's replication scheme adopts a primary-backup with the host that is being replicated acting as a primary, and other replicas as the backups. And a voting protocol within a Nysiad's replicated state machine (RSM) of host ensures that the output of the RSM is valid.

The Nysiad system distinguishes two kinds of systems, *original* system and *translated* system, which refer to the systems before and after translation, respectively. The original system tolerates only crash failures, while the translated system tolerates Byzantine failures as well. Also, Nysiad assumes that each host runs a deterministic state machine that transitions in response to receiving messages or expiring timers. Besides, the Nysiad system works under an asynchronous environment. The Nysiad transformation requires a "guard graph", which is a communication graph to establish a connection between hosts. We can literally consider each guard as a *backup* on the primary-backup scenario of RSM. And, a $t-$guard graph means for each host at most t of its guards are Byzantine, and Nysiad works with guard graph to perform translation. With this regards, Nysiad translates the original system by replicating the deterministic state machine of a host onto its neighbor guards. In general, a Nysiad instance includes four sub-protocols. The *replication protocol* ensures that guards of a host remain synchronized. The *attestation protocol* guarantees that messages delivered to guards are messages produced by a valid execution of the protocol. The *credit protocol* forces a host to either process fall its input fairly, or to ignore all input. And, the *epoch protocol* allows the guard graph to be bootstrapped and reconfigured in response to host churn.

However, there exist some potential equivocation in Nysiad, in which the most may send different messages to different guards or order its messages differently for different guards. To handle this equivocation, the guards gossip among each other to agree on the order and content of messages sent by the host. And the communication complexity of the adopted gossip is quadratic in terms of guards. Besides, Nysiad can handle non-deterministic state machines, however, doing so requires protocol changes to treat non-deterministic events as inputs [157]. For details of design principles, interested readers can refer to Section 4 in [189].

A similar and earlier work on this type of RSM for Byzantine failure is the PeerReview, which was proposed by Haeberlen et al. in 2007 [190]. PeerReview is a system that employs witnesses to collect a tamper-evident record of all messages in a distributed system for subsequence checking against a reference implementation. Essentially, PeerReview provides accountability [191] in distributed systems, which ensures that Byzantine faults whose effects are observed by a correct node are eventually detected and irrefutably linked to a faulty node. Also, the peer review process can ensure that a correct node can always defend itself against false accusations. PeerReview assumes that each host implements a protocol using a deterministic state machine. It works by maintaining a secure record of the incoming and outgoing messages by each node, and periodically, runs logs through the state machines, and checks output against outgoing logs. It can only detect a subclass of Byzantine failures, and only after the fact.

However, PeerReview does not provide fault tolerance, instead, it provides eventual fault detection and localization, which the system's designers argue leads to fault deterrence. Its tamper-evident record is a distributed hash chain, which is used to detect equivocation about the messages recorded in the log. Besides, the communication required to collectively manage the tamper-evident message log is quadratic in the size of the witness set.

11) BFTSim - (architecture): BFTSim is a simulation tool to Byzantine fault-tolerant protocols proposed by Singh et al. in 2008 [63]. It equips with a declarative networking system with a robust network simulator for various protocols. Protocols can be rapidly implemented from pseudocode in a high-level declarative language, and the network conditions and measured costs of communication packages and crypto primitives can be plugged into the latter. In general, BFTSim can be used as a fast prototype module to predict the performance of designed protocols before practical implementation. As a simulation framework, BFTSim couples a high-level protocol specification language and an execution system with a computation-aware network simulator built atop $ns-2$ [192]. This allows the users to rapidly implement protocols based on pseudocode descriptions, evaluate their performance under a variety of conditions, and isolate the effects of implementation artifacts on the core performance of each protocol. Fig. 24 shows an abstract of the BFTSim software architecture. Interested readers can refer to the work [63] for more details. From the BFTSim open source tool online (http://bftsim.mpi-sws.org/), it seems, from 2009, these source codes were out of maintenance.

12) Byzantium - (middleware): Byzantium is a Byzantine fault-tolerant database replication middleware that provides snapshot isolation (SI) semantics, proposed by Preguica et al. in [193]. Like commit barrier scheduling (CBS) in HRDB [184], Byzantium is a kind of middleware to enhance concurrency. Byzantium improves on the existing BFT replication schemes for database, in which it has no centralized components, of whose correctness the integrity of the system depends; and it allows an increased concurrency to achieve good performance. Fig. 25 shows an abstract of system architecture. Under Byzantium, it uses the PBFT state machine replication algorithm as its consensus protocol, and follows the system models of PBFT, e.g., using $3f + 1$ replicas to tolerate f faulty replicas and asynchronous communication model. And SI is a weaker form of semantics that is supported by most commercial databases, which allows a transaction to logically execute in a database snapshot.

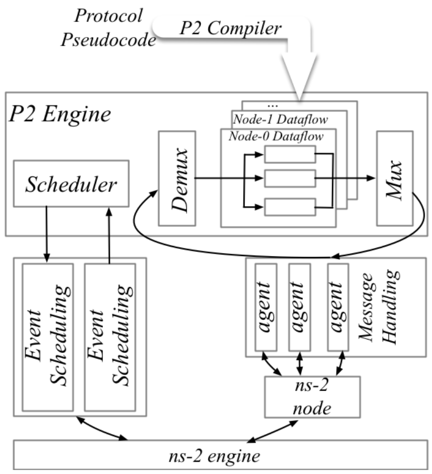


Fig. 24. The BFTSim software architecture [63].

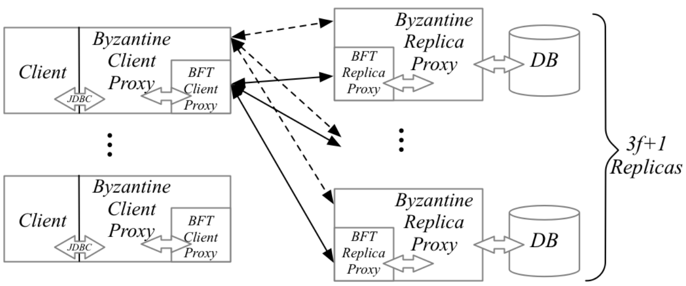


Fig. 25. Byzantium system architecture [193].

Only when there is no write-write conflict with any committed concurrent transaction, a transaction can commit, otherwise, it must abort. Compared with serializability in replicated machine systems, SI can increase concurrency among transactions. This is because, with the SI, only write-write conflicts must be avoided, which is beneficial for read-only transactions without needing to block or to abort them. Besides, Byzantium as a middleware can be used with off-the-shelf database systems and builds on top of an existing BFT library.

13)Zeno - (feature): Zeno is a BFT state machine replication protocol that trades consistency for higher availability, proposed by Singh et al. in 2009 [194]. Most proposals for Byzantine fault-tolerant protocol have focused on strong semantics, such as linearizability [195], where the replicated systems appear to the clients as a single, correct, and sequential server. With the popularity of data centers and cloud services, ensuring correct and continuous operation of these services is critical, which requires a higher availability. Zeno is designed to meet the needs of modern services running in corporate data centers. More preciously, Zeno favors service performance and availability, at the cost of providing weaker consistency guarantees than traditional BFT replications when network partitions and other infrequent events reduce the availability of individual servers. Still, Zeno offers eventual consistency semantics [196], in which different clients can be unaware of the effects of each other's operations, e.g., during a network partition, but operations are never lost and will eventually appear in a linear history of the service once enough connectivity is re-established. The authors also describe the necessaries and sufficiency of the eventual consistency from the standpoint of many applications. For example, when a network partition occurs, eventual consistency is necessary to offer high availability to clients on both sides of the partitions.

Zeno adapted Zyzzyva to provide availability mainly because Zyzzyva explores the speculation to conclude operations fast and cheaply, having an ability to yield high service throughout during favorable system conditions. As a BFT replication protocol, Zeno requires $3f + 1$ replicas to tolerate f faults, with an arbitrary number of Byzantine clients. Based on the baseline Zyzzyva (a hybrid agreement/quorum model), it made some modifications on the levels of consistency. For example, Zeno distinguishes two kinds of quorums: *strong quorums* consisting of any group of $2f + 1$ distinct replicas, and *weak quorums* of $f+1$ distinct replicas. Like most traditional BFT protocols, Zeno has three components: *sequence number assignment* to determine the total order of operations, *view change* to handle leader replica election, and *checkpointing* to deal with garbage collection of protocol and application state. We recommend interested readers to read Section 4 of [194] on each operation.

In general, a weak consistency allows replicas to temporarily diverge and users may see inconsistent data, however, with better availability and performance. There are some trade-offs on choosing between consistency and availability. Different applications may have different choices. For example, a cluster of distributed data centers may favor availability, while a decentralized distributed ledger system may instead favor consistency. However, to property of consistency itself, for some permissionless blockchain settings (e.g., PoW as a consensus), it temporarily allows a certain level of diverging (in the form of forks), however, an eventual consistency must be maintained. While for a BFT based blockchain system, a strong consistency (or linearizability) is more critical.

14)Aardvark - (architecture): Aardvark is a BFT system designed to be robust to failures, proposed by Clement in 2009 [153]. It criticizes the existing BFT (before 2009) state machine replication protocols, although fast, did not tolerate Byzantine faults very well. For example, even a single faulty client or server is capable of rendering PBFT, Q/U, HQ, and Zyzzyva, and make them virtually unusable. For those BFTs, the complexity often undermines robustness in two ways: 1) the protocols' design include *fragile optimizations* that allow a faulty client and server to knock the system off of the optimized execution path to an expensive alternative path, and 2) the protocols' implementation often fails to handle properly all of the intricate corner cases. Based on the above observations, the authors advocate a new approach to build *robust BFT* (RBFT) systems and avoid performing fragile optimizations. The goal of RBFT is to shift the focus from constructing high-strong systems that maximize best-case performance to constructing systems that offer acceptable and predictable performance under the broadest possible set of circumstances. Aardvark was based on a new design philosophy to provide a robust BFT replication system. Essentially, Aardvark itself was built on the classic PBFT protocol under a practical asynchronous network where *synchronous intervals*, during which messages are delivered with a bounded delay, occur infinitely often.

The authors define a set of principles for constructing BFT services that remain useful even when Byzantine faults occur. Specially, the temptation of fragile optimizations should be avoided in a practical design, and a BFT system should be designed around an execution path that meets some properties. More precisely, a BFT system should provide acceptable performance; a BFT system should be easy to implement; and a BFT system should be robust against Byzantine attempts to push the system away from it. Also, optimizations for the common case should be acceptable only if they do not endanger these properties. Due to the FLP impossibility, the liveness of protocol cannot be guaranteed in an asynchronous environment. Even it cannot achieve liveness, a BFT protocol should perform well under a weak assumption, e.g., *uncivil executions*. An execution is uncivil *iff* (a) the execution is synchronous with some implementation-dependent short bound on message delay, (b) up to f servers and an arbitrary number of clients are Byzantine, and (c) all remaining clients and servers are correct.

Aardvark is based on the above design philosophy and a weak assumption on uncivil executions. The Aardvark protocol essentially consists of three stages: client request transmission, replica agreement, and primary view change. And there are three main design differences between Aardvark and previous BFT systems, namely, signed client requests, resource isolation, and regular view change. For signed client requests, Aardvark clients use digital signatures to authenticates their requests. Digital signatures can provide non-repudiation and ensure all correct replicas make identical decisions about the validity of each client request. Aardvark uses them *only* for client requests which push the expensive generation of signature onto the client and leave the servers with less expensive verification operations. Meanwhile, the communication on primary-to-replica, replica-to-replica, and replica-to-client rely the MAC authentication scheme (a fast and less expensive operation). For resource isolation, the Aardvark prototype implementation explicitly isolates network and computational resources. For example, implementing replica-to-replica communication reduces the potential hazardous cases on message transmission. For regular view changes, the Aardvark protocol invokes view change operations on a regular basis to prevent a primary from achieving tenure and exerting absolute control on system throughput. For example, Aardvark monitors the performance of the primary and changes the view in case it seems to be performing slowly. Based on the above optimization on Aardvark, the work [153] also presented detailed implementation and evaluation on Aardvark. Interested readers can look into the details.

15) Client Speculation - (middleware): Client speculation is a technique to perform a speculative execution at the clients to reduce the impact of network and protocol latency, proposed by Wester et al. in 2009 [197]. The features of the geographical distribution of replicas in an SMR system increase the network latency between replicas, and many applications and protocols for SMR are highly sensitive to latency. For example, in traditional Byzantine fault-tolerant protocols, it must wait for multiple replicas to reply, and the effective latency of services is limited by the latency of the slowest replica (even an honest one) being waited for. Agreement-based protocols, e.g., PBFT, typically require multiple message rounds to reach an agreement which further exacerbates the user experience (e.g., high network latency). Client speculative execution allows clients of replicated services to be less sensitive to high latencies caused by network delays and protocol messages. These phenomena are more clear in the case that faults are generally rare or in the absence of faults, and the response from even a single replica is an excellent predictor of the final. Based on these observations, the clients in client speculation can proceed after receiving the first response, thereby hiding considerable latency in the common case in which the first response is correct. And this will clear if at least one replica is located nearby. To ensure the safety in the case where the first response is incorrect, a client may only continue executing in the way *speculatively*, until enough responses are collected to confirm the prediction. Besides, client speculation hides the latency of the replicated service from the client, and replicated servers can optimize their behavior to maximize their throughput and minimize load, e.g., by handling agreement in large batches.

To combine the client speculation technique into existing replication protocols, some modifications are required on these protocols. The authors provide some design principles for using client speculation with replicated services, e.g., generating early replies, prioritizing throughput over latency, avoiding speculative state on replicas, and using replica-resolved speculation. For generating early replies, to minimize the latency, a protocol should be designed to get the first reply to the client as quickly as possible, in which the fastest reply can be from the closest replica and that replica responds immediately. Thus, a client speculation-supported protocol should have one or more replicas immediately respond to a client with the replica's best tentative result (or guess) for the final outcome of the operation. Due to the nature of the guess, that reply is not guaranteed to be correct in the presence of a malicious responded replica. For instance, for agreement-based protocols like PBFT, a more practical way is to have only the primary execute the request early and respond to the client, and such predictions are correct unless the primary is faulty. However, this way presumably assumed that the primary is located near the client to get minimized latency. For prioritizing throughput over latency, when there is a case that needs to make a trade-off between throughput and latency, a better choice is to consider the one that improves throughout over one that improves latency (even under client speculation). This is because the speculation can do much to hide replica latency but little to improve replica throughput. Even the client can get an early response, however, the final outcome of an operation depends on the latency of the overall operations among the replicas. For avoiding a speculative state on replicas, speculative execution must avoid *output commits* that externalize speculative output to ensure correctness, as such output cannot be undone once externalized. To avoid the inconsistency of replicas, the authors recommend selecting the smallest boundary of speculation, which disallows replicas from storing the speculative state. For using replica-resolved speculation, even with the smallest boundary of speculation, the clients are still allowed to issue new requests that depend on the speculative state (or called *speculative requests*). These requests can be handled concurrently, increasing throughput when the replicas are not already fully saturated. However, there is no mechanism for a replica to determine whether or not a client received a correct speculative response. Interested readers can refer to Section 2 of [197] for more detailed design principles.

Besides, following these mentioned design protocols, the authors presented a prototype of a modified protocol, PBFT-CS, based on PBFT protocol, and applied the modified protocol on replicated NFS and counter services. Evaluation results show that client speculation trades in 18% maximum throughput to decrease the effective latency under light workloads, which speed up to run time $1.08 - 19X$ on the case of a single-client co-located with the primary.

16)UpRight - (feature): UpRight is a state replication system and provides a library for fault-tolerant services, proposed by Clement et al. in 2009 [198]. The UpRight library seeks to make Byzantine fault tolerance a simple and viable alternative to crash fault tolerance for a range of cluster services. On the design choices, UpRight favors simplifying adoption by existing applications, and performance is a secondary concern. To make a BFT system competitive with the CFT system, it does not only enhance the optimizations on performance, hardware overhead, and availability, but also in terms of engineering efforts. The UpRight draws heavily to retain performance, hardware overhead, and availability; however, it favors minimizing intrusiveness

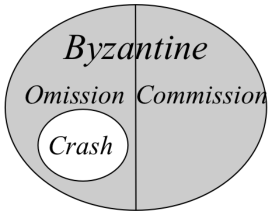


Fig. 26. Failure hierarchy. Byzantine failures include omission failures and commission failures. Crash failures are a subset of omission failures [198].

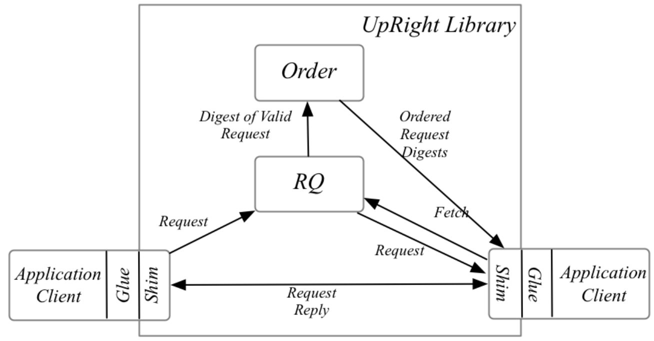


Fig. 27. UpRight architecture [198].

to existing applications over the raw performance. By applying the UpRight library, the authors construct UpRight-Zookeeper and UpRight-HDFS using open-source codes based on the Zookeeper [54] and HDFS (Hadoop Distributed File System) [199]. Both UpRight-based systems provide improvements in fault tolerance. For example, the original HDFS system can be halted by a single fail-stop node, while UpRight-HDFS has no single point of failures and also provides end-to-end Byzantine fault tolerance against faulty clients, DataNodes, and NameNodes. UpRight makes standard assumptions for Byzantine fault-tolerant systems with few modifications. For example, the safety of UpRight holds in any asynchronous distributed systems, however, the liveness is guaranteed only during *synchronous intervals* in which messages sent between correct nodes are processed within some fixed (but potentially unknown) worst-case delay from when they are sent.

UpRight distinguishes two kinds of Byzantine failures: *Omission* failures (including crash failures) in which a node fails to send one or more messages specified by the protocol and sends no incorrect messages (or nothing) based on the protocols and its inputs, and *commission* failures, which include all failures that are not omission failures, including all failures, e.g., a node sends a message that is not specified by the protocol. Fig. 26 shows a failure hierarchy. And UpRight can provide the following properties: 1) an UpRight system is safe despite r commission failures and any numbers of omissions failures; 2) an UpRight system is safe and eventual live ("up") during sufficiently long synchronous intervals when there are at most u failures of which at most r are commission failures and the rest are omission failures, where $u >= r$. For example, when $u = r$, the system is equivalent to the one based on a full Byzantine failure model, and when $r = 0$, the system is equivalent to the one based on a crash failure model. UpRight implements state machine replication, which tries to isolate applications from the details of the underlying replication protocol. This kind of design principle makes it easy to convert a CFT application into a BFT one or to construct a new fault-tolerant application. Essentially, the UpRight library should ensure some features, e.g., each application server replica sees the same sequence of requests and maintains a consistent state, and an application client sees responses consistent with this sequence and state. To achieve these features, the applications must address some specific challenges in request execution and checkpoint management. For the request execution, applications must account for non-determinism, multi-threaded execution, read-only requests, and spontaneous server-initiated replies. For the checkpoint management, the checkpoints should be inexpensive to generate, inexpensive to apply, deterministic, and nonintrusive on the codebase.

Fig. 27 shows an UpRight's high-level architecture. It is important to describe how it works for cluster applications. As the figure shown, UpRight requires three modules to execute a client request: request quorum (RQ), order, and execution. An UpRight client deposits its request at the request quorum, which stores the request, forwards a digest of the request to the order module, and supplies the full request to the execution module. The order module produces a totally ordered sequence of batches of request digests. The execution module embodies the application's server, which executes requests from the ordered batches and produces replies. Although UpRight's core is a Byzantine agreement protocol, it departs from prior BFT protocols in some aspects. In general, the RQ stage of UpRight helps avoiding complex corner cases, and avoids sending large requests through the order state. The agreement protocol combines ideas from three prior replication systems, i.e., Zyzzyva's speculative execution [59], Aardvark's techniques on robustness [153], and Yin et al.'s techniques for separating agreement and execution [176]. Besides, it supports a parameterized and easy configuration to minimize replication costs. For example, the UpRight prototype allows its users to separately configure u (the number of failures it can tolerate while remaining up) and r (the number of failures it can tolerate while remaining right). Interested readers can refer to the work [198] for more detailed information.

17)Spinning - (feature): Spinning is a BFT algorithm to mitigate performance attacks by changing the primary, proposed by Veronese et al. in 2009 [200]. Most leader-based Byzantine fault-tolerant replication systems have a *primary* replica that is in charge of ordering the clients' requests. However, it was shown that this kind of dependence allows a faulty primary has the ability to degrade the performance of the system to a small fraction of what the environment allows (e.g., PBFT's performance attacks proposed in Prime [187]). In performance attacks, it generally has two types: pre-prepare delays (an attack to delay the ordering of the requests) and timeout manipulations (to increase the timeouts used in BFT). For example, in a pre-prepare delay attack, the primary imposes a maximum delay on the execution of requests, but only on the first request of a queue of pending requests, so a faulty primary can process one request at a time, strongly delaying most requests. These two types of attacks can also apply to some of the algorithms derived from PBFT. According to the description of the Spinning algorithm, which modifies the usual form of operations of PBFT, instead of changing the primary when it is suspected of being faulty, it changes the primary whenever it defines the order of a single batch of requests. Simply, in each view the primary orders only one batch. As views are always changing, it has no view change operation, instead, it has a *merge* operation. The merge operation is in charge of putting together the information from different servers to decide if requests in views that "went wrong" are to be executed or not.

Even with a Spinning algorithm, a faulty replica still has a chance to be the primary to impact the average performance of services. To solve this issue, Spinning algorithms use a punishing misbehavior mechanism to punish the primary when something goes wrong. For example, basically putting them in a blacklist, a server in the blacklist does not become the primary. Spinning primary has some benefits. One benefit is that it can avoid the above-mentioned performance attacks made by faulty primaries in a very simple and efficient way, and the view change in Spinning does not occur. The view is changed automatically after the three communication steps are executed by the servers. Another benefit is that Spinning can improve the throughput of PBFT when there are no faulty servers by balancing the load of ordering requests among all correct servers. Intuitively, ordering requests requires that all servers exchange messages, causing load in all of them, however, most load is in the primary so changing the primary improves the throughput of the algorithm by a factor of 20% (some similar work by Mao et al. in Mencius [201]).

The authors also designed a prototype based on PBFT under a partial synchrony system model with $3f + 1$ replicas to tolerate f faulty replicas. The communication pattern of the Spinning algorithm almost follows the PBFT to reach an agreement. Due to no view change sub-protocol of in PBFT, instead, Spinning resorts to a merge operation, which has the similar objective to ensure liveness. To short, the algorithm has to *merge* the information from the different servers to agree on the requests that were accepted and go to the next view. Also, the Spinning algorithm defines some mechanisms to punish the misbehavior of faulty primaries. Besides, the authors also presented several optimizations to the basic Spinning algorithm, including batches of requests, piggybacking *pre-prepare* messages, using MAC vectors, and parallel executions. Interested readers can refer to Section III-F of [200] for more details.

18) Zzyzx - (feature): Zzyzx is a scalable Byzantine fault-tolerant protocol through *Byzantine Locking*, proposed by Hendricks et al. in 2010 [202]. By using a *Byzantine Locking* scheme, Zzyzx allows a client to extract state from an underlying replicated state machine and access it via a second protocol specialized for use by a single client. The second protocol requires just one round trip and $2f+1$ responsive servers. And the extracted state can be transferred to other servers, allowing non-overlapping sets of servers to manage different states. This enables Zzyzx's throughput to be scaled by adding servers *when concurrent data sharing is not common*. Practically, it can perform well and achieves scalability in the case that faults are rare and the concurrency is uncommon (aka. a benign environment). However, when the concurrency is common, Zzyzx performs similarly to its underlying protocol (e.g., PBFT). Zzyzx takes inspiration from the locking mechanism used by many distributed systems to achieve high performance in a benign environment. For example, the metadata service of most distributed file systems contains a distinct object for each file or directory, and the case of concurrent sharing is rare [203] [204].

Byzantine Locking is layered atop on BFT consensus protocol, and it *temporarily* gives a client exclusive access to state in a replicated state machine, which has the ability to extract the relevant state of the underlying replicated state machine. In general, Byzantine Locking is only a performance tool. To ensure the liveness of a replicated system, a locked state is kept on servers, and a client that tries to access objects locked by another client can request that the locks be revoked, forcing both clients back to the underlying replicated state machine to ensure consistency. Zzyzx uses $3f + 1$ servers to tolerate up to f Byzantine servers, and a physical server can take different roles, e.g., *log servers* or state machine *replicas*. Byzantine Locking provides a client an efficient mechanism to modify replicated objects by providing the client temporary exclusive access to the object. A client that holds temporary exclusive access to an object is said to have locked the object. Simply, the execution of Zzyzx can be divided into three interfaces/sub-protocols: *substrate interface*, *log interface*, and *unlock sub-protocol*. If a client has not locked the objects needed for an operation, the client uses a substrate interface protocol, such as PBFT or Zyzzyva. If a client holds locks for all objects touched by an operation, the client uses the lock interface. If a client tries to access an object for which another client holds a lock, then the unlock sub-protocol is been effective. For more details on each operation, interested readers can refer to Section 4 in [202].

19) Generic Consensus Algorithm - (architecture): Generic consensus algorithm is not a specific consensus algorithm, instead highlighting the basic and common features of known consensus algorithms, proposed by Rutti et al. in 2010 [205]. It provides a generic framework for the building of consensus algorithms, which is a parameterized formation of the algorithm. Numerous consensus algorithms had been proposed with different features and different fault models. Considering these numerous algorithms, a classification, e.g., via parameter forms, would be helpful to identify the basic mechanisms on which they rely. The parameters of the generic algorithm encapsulate the core differences between various consensus algorithms, including leader-based and leader-free algorithms, addressing benign faults, authenticated Byzantine faults, and Byzantine faults. The generic consensus algorithm consists of some successive phases where each phase is composed of three rounds: a selection round, a validation round, and a decision round. Some existing algorithms may skip the validation round, which introduces some dichotomy among consensus algorithms, the ones that require the validation round, and the others for which the validation round is not necessary.

In general, the proposed generic algorithm is based on four parameters: the *FLV* function, the *Selector* function, the threshold parameters, and the flag *FLAG*. The functions FLV and Selector are characterized by abstract properties; the threshold parameter is defined with respect to n (the number of processes), f (the maximum number of benign faults), and b (maximum number of Byzantine processes). The authors proved the correctness of the generic consensus algorithm by referring only to the abstract properties of the above-mentioned parameters. Also, the authors argued that the correctness proof of any specific instantiated consensus algorithm consists simply in proving that instantiations satisfy the abstract properties of the corresponding functions. The generic consensus algorithm provides more detailed and elaborated classifications, e.g., on the honesty of nodes. The *honest* processes can be differentiated into *correct* or *faulty*, where an honest process is faulty if it eventually crashes, and is correct otherwise. Also, it assumes that among n processes, there exist at most b Byzantine processes and at most f faulty (honest) processes. Besides, it distinguishes different Byzantine faults in literature: *authenticated Byzantine* faults, where the communicated messages are signed by their sending process (with the assumption that signatures cannot be forged by any other process), and *Byzantine* faults, where there is no mechanism for signatures (but the receiver of a message knows the identity of the sender). Also, the authors discussed the way to adopt to support randomized consensus algorithms. Interested readers on the framework and instantiations of the generic consensus algorithm can refer to the work [205].

Besides the above generic consensus algorithm, there exist several literatures on this topic. For example, Mostéfaoui et al. [206] proposed a consensus framework restricted to benign faults, which allows unification of leader oracle, random oracle, and failure detector oracle. Gueraoui and Raynal [207] also proposed a generic consensus algorithm, where its generality is encapsulated in a function *Lambda*, which further encapsulates both selection and validation rounds; whose later version on *Omega* encapsulates communication abstraction. And Song et al. [208] proposed some key building for building blocks replicated systems and constructing consensus algorithms.

20)Breaking the $O(n^2)$ Bit Barrier - (theory): This work presents a scalable Byzantine agreement for an adaptive adversary, proposed by King and Saia in 2010 [209]. It focuses on the theoretical perspective to improve the scalability under Byzantine replicas and the adaptive adversary. In general, the proposed algorithm is scalable in the sense that each processor sends only $\tilde{O}(\sqrt{n})$ bits, where n is the total number of processors. It also works with a high probability against an *adaptive adversary*, which can take over processors at any time during the protocol, up to the point of taking over arbitrarily close to a $1/3$ fraction of the total replicas. The system model is based on synchronous commutation but with a rushing adversary. More precisely, the algorithm introduces only $\tilde{O}(\sqrt{n})$ bits processor overhead, which leads to it with $\tilde{O}(\sqrt{n})$ bit complexity. And the communication with $\tilde{O}(\sqrt{n})$ bit complexity can be considered as a distributed version of a bit-fixing random source. Due to the randomness, it still has a small probability of errors. Based on the observations, the authors claim that any algorithm that uses cryptography is subject to errors with a certain probability, since the adversary may get lucky and break the cryptography.

The proposed consensus algorithm is assumed to build on a network with *private* and unauthenticated peer-to-peer communication channels. For example, whenever a processor sends a message directly to another, the identity of the sender is known to the recipient without resorting to cryptographic assumptions. An adaptive adversary is not only can take over processors at any point during the protocol, but also can learn the processor's state, so that the bad processors can engage in any kind of deviation from the protocol. Also, these bad processors can send any number of messages, a kind of flooding attack. On the network synchrony assumption, the proposed algorithm works under a synchronous model with the *rushing* adversary. In a rushing adversary, the bed processors may wait to receive all messages sent by the good processors before they need to send out their own messages. Many theoretical results and proofs are presented in the paper, and interested readers can read the paper [209] for more details.

21)Abstract - (architecture): *Abstract* is an acronym of Abortable Byzantine faulT toleRant stAte maChine replicaTion, an abstraction to reduce the development cost of BFT protocols, proposed by Guerraoui et al. in 2010 [14]. In the work of *Abstract*, it shows another two BFT protocols: *AZyzzyva*, a protocol that mimics the behavior of Zyzzyva in the best-case scenario, and *Aliph*, a BFT protocol that outperforms previous (before the year of 2010) BFT protocols both in terms of latency and throughput. *Abstract* treats each BFT protocol as a composition of instances of the *Abstract* framework and each instance can be developed and analyzed independently. When designing and implementing an SMR protocol, two objectives typically are very crucial: robustness and performance. Robustness shows the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures and asynchrony. While, performance measures the time it takes to respond to a request (latency) and the number of requests that can be processed per time unit (throughput). With various models and settings for consideration in replicated systems, many of those designed Byzantine protocols are notoriously difficult to develop, test, and prove. Even under the best-case scenario, there is no "one size fits all" BFT protocol [63]. For example, the performance differences among the protocols can be heavily impacted by the actual networks, the size of the messages, the vary nature of the "common" cases, etc. One can design new protocols outperforming all others under specific circumstances, however, it is hard to evaluate performance with previously designed protocols in different circumstances. The *Abstract* provides an abstraction to reduce the deployment cost of BFT protocols.

Following the divide-and-conquer principle, the *Abstract* views BFT protocols as a composition of an instance of its framework, and each instance can target and optimize for specific system conditions. In general, an instance of *Abstract* looks like BFT SMR, with only one exception that it may sometimes abort a client's request. We can consider a BFT protocol as a composition of instances of *Abstract*, and each instance itself is a protocol that commits clients' requests except if certain conditions are not satisfied (e.g., it can abort requests). This provides a way of composability, as well as flexibility, for the design of BFT protocols. More specifically, the composition of any two *Abstract* instances is *idempotent*, yielding yet another *Abstract* instance. Thus, the processes of developing a BFT protocol (e.g., design, test, proof, and implementation) can be simplified into two easy-going tasks: developing individual *Abstract* instances and ensuring that a request is not aborted by all instances. The first task is typically much simpler than developing a full-fledged BFT protocol and allows for very effective schemes. A single *Abstract* instance can be crafted solely with its progress in mind, irrespective of other instances. The second task can also be easily achieved via *Abstract*, for example, by simply reusing, as a black-box, an existing BFT protocol as one of its instances, without indulging in some complex modifications. This requires that the designer of a BFT protocol only has to ensure two things. One is that individual *Abstract* implementations are correct, respectively of each other, and the other is that the composition of the chosen instances is live, e.g., that every request will eventually be committed. *AZyzzyva* and *Aliph* provide two examples on the above design principles and implementations.

AZyzzyva is a protocol to illustrate the ability of *Abstract* to significantly easing the design, implementation, and proof of BFT protocols. It is a full-fledged BFT protocol to mimic Zyzzyva in its "common case", and it can be considered as a composition of two *Abstract* instances: *ZLight* and *Backup*. Note that the "common case" also called "best-case", e.g., when there are no link or server failures, relying on *ZLight*; while the "other cases", e.g., the periods with asynchrony/failures, rely on *Backup*. Roughly speaking, *ZLight* is an *Abstract* instance that guarantee progress in the Zyzzyva's "common case", while *Backup* is an *Abstract* instance with strong progress: it guarantees to commit an exact certain number of requests k (k is itself configurable) before it starts aborting. The *Backup* instance can be as a black-box to handle a legacy BFT protocol. More precisely, *Backup* works as follows: it ignores all requests delivered by the underlying BFT protocol until it receives a request containing a valid *init* history, e.g., an unforgeable abort history generated by the preceding *Abstract* (*ZLight* in the case of *AZyzzyva*). Besides, the code line count and the proof size required to obtain the AZyzzyva are conservatively less than $1/4$ those of Zyzzyva.

Aliph is a protocol to demonstrate the ability of *Abstract* to develop efficient BFT protocols. Along with the *Backup* instance used in *AZyzzyva*, *Apliph* uses two other new instances: *Quorum* and *Chain*. A *Quorum* instance commits requests in an ideal case, e.g., the case that there are no server/line failures, no client Byzantine failures, and no contention. In general, the *Quorum* implements a very simple communication pattern and gives *Aliph* the low latency flavor when its progress conditions are satisfied. It requires only one round-trip of message exchange between client and replicas to commit a request, in which the client sends the request to all replicas that speculatively execute the request and send a reply to the client. While the *Chain* provides exactly the same progress guarantee as *ZLight*, e.g., it commits requests as long as there are no server/link failures or Byzantine clients. *Chain* implements a pipeline pattern (with all replicas knowing the fixed ordering of replicas' IDs and a novel authentication technique), and allows *Aliph* to achieve better peak throughput than all existing protocols (before the year of 2010). Also, *Aliph* uses the following static switching orderings to orchestrate its underlying protocols: *Quorum-Chain-Backup-Quorum-Chain-Backup-etc*. Besides, each of *Quorum* and *Chain* can be developed independently and requires much less of the code needed to develop the previous BFT protocols.

Interested readers on the details of constructions of *Abstract*, *AZyzzyva*, and *Aliph* can refer to the work [14].

22) Byzantizing Paxos - (feature): Byzantizing Paxos presents a process to derive Byzantine Paxos consensus algorithm by the so-called *Byzantizing*, proposed by Lamport in 2011 [210]. Specially, the process derives a $3f+1$ Byzantine Paxos consensus protocol by *Byzantizing* a variant of the ordinary Paxos algorithm. This process is achieved by having $2f+1$ non-faulty processes to emulate the ordinary Paxos algorithm despite in the presence of f malicious processes. Besides, the author also presented a formal, machine-checked proof that the Byzanitized algorithm can implement the ordinary Paxos consensus algorithm under a suitable refinement mapping scenario.

In more details, the Paxos algorithm typically uses $2f+1$ processes to tolerate the *benign* failure of any f of them. While, the Byzantine algorithm, e.g., PBFT, uses $3f+1$ processes to tolerate f Byzantine (malicious faulty) processes. Strictly, they are different fault-tolerant algorithms, and the Byzantine algorithm is not a Byzantine version of Paxos algorithms even both algorithms can be used to tolerate failures. Also, there already existed some attempts, from the aspect of *abstract and non-distributed* algorithm, to explain the relation between a Byzantine Paxos algorithm and an ordinal Paxos algorithm in [211]. The work proposed in [210] took a direct approach and derived a Byzantine Paxos algorithm from a *distributed non-Byzantine* one by a procedure of *Byzantizing*. The Byzantizing procedure converts an N-process algorithm that tolerates the benign failure of up to f processes into an $(N+f)$ process algorithm that tolerates f Byzantine processes. In the Byzantine algorithm, the N good processes emulate the execution of the original algorithm despite in the presence of f Byzantine ones.

The author presented a detailed process to Byzantine a variant of the classic Paxos consensus algorithm, called *PCon*, and a detailed process to Byzantine an abstract generalization of the Castro-Liskov Byzantine consensus algorithms (i.e., PBFT), called *BPCon*. More specifically, BPCon is derived from PCon via a TLA$^+$ theorem [212], which asserts that BPCon implements PCon under a suitable refinement mapping [213]. Besides, the author argued that other Byzantine Paxos consensus algorithms can also be derived by Byzantizing versions of Paxos. Interested readers can refer to the work [210] for more details.

23)Obfuscated BFT (OBFT) - (feature): Obfuscated BFT (OBFT) explores the idea of *obfuscation* in a BFT context, which is currently a technical report by Shoker et al. in 2011 [214]. In OBFT, with the honest but possible crash-prone clients, the replicas remain unaware of each other to achieve obfuscation property by avoiding any direct inter-replica communication. Classic BFT protocols rely on inter-replica communication to ensure one-copy semantics, which makes these protocols fragile. In that scenario, the replicas must share some access information about each other, where a distributed DoS attack may threaten the system security by accessing those shared access information. By avoiding any direct inter-replica communication, no replica knows anything about the others. In this case, the client plays a crucial role in OBFT, and OBFT assumes that clients cannot be malicious though they can fail by crashing. Practically, if replicas do not communicate, a malicious client can easily violate consistency. Even under the fact that clients are trusted, OBFT still faces several challenges. A good obfuscating BFT protocol should handle some critical scenarios and meet some criteria. For example, clients can still crash. If the corresponding obfuscated BFT does not tolerate a crashed client, the unique request ordering among replicas can be compromised by the other clients. Upon failures detected, a recovery is needed. Besides, the obfuscated BFT should handle contending clients that can force the copies of an object on different replicas to skew.

In general, the proposed OBFT requires $3f+1$ replicas to tolerate f Byzantine faults, and the client in OBFT communicates with $2f+1$ *Active* replicas in its *Speculative* phase. OBFT assumes that clients may fail by crashing, but they do not behave maliciously, and the liveness is guaranteed by the assumption of eventually synchronous. Typically, using $2f+1$ replicas only at a time can sustain faults but cannot ensure the progress. Thus, OBFT launches a *Speculative* phase on $2f+1$ *Active* replicas. Upon failures detected, it recovers by replacing the *Suspicious* replicas with some correct replicas from the f *Passive* ones, and then resumes to the speculative phase on a new *Active* set in a new view. At any time, the protocol distributes the replicas over three sets: Active set, Passive set, and Suspicious set. The Active set consists of $2f+1$ replicas or active replicas, which are used in the speculative phase; the Passive set consists of f idle replicas that are used as the recovery backups; and the Suspicious set is a virtual set, consisting of up to f replicas that are either faulty or slow, which comprises the Passive replicas during the speculative phase. Upon the finish of failure detection, i.e., in *recovery* phase, the client identifies f Suspicious replicas, and replaces them with another f replicas, possibly from the Passive set or from the entire $3f+1$ replicas.

The OBFT algorithm consists of two main phases: a speculative phase and a recovery phase. For the speculative phase, its communication pattern in a failure-free scenario is simple, which is concerned only with the Active set. For the recovery phase, it takes place using both Passive and Active sets. During the speculative phase, OBFT makes some optimizations compared to the classic speculative protocols, such as Zyzzyva. For instance, OBFT pushes multi-cast and MAC overhead towards to its client, and keeps a light primary. Excluding the primary, all replicas validate the order of clients' requests upon their receipts. To maintain failure independence, obfuscation aspects require replicas to be unknown and unaware of each other. Replicas usually need to communicate for two purposes: ensure the atomic request execution and validate the correctness of response. However, the final decision is made by the client. The recovery phase is composed of three major steps: aborting, collecting abort history, and cleaning the Active set from any suspicious replicas. Although much detailed information still needs to be filled until the time we reviewed this technique report, interested readers can refer the more details on the description of the protocol.

Besides, the authors state that OBFT is designed to be deployed on WANs, and preferably cloud of distinct providers, different platforms, and located in geographically distinct locations around the universe. Also, OBFT achieves a certain level of scalability compared with the prior protocols. For example, OBFT reduces the load on replicas by pushing multicast and expensive cryptographic operations off the clients, and it imposes an almost identical load on all replicas (including the primary).

24) ZZ - (feature): ZZ is an approach to reduce the replication cost of BFT services from $2f+1$ to practically $f+1$, proposed by Wood et al. in 2011 [215]. From a high-level perspective, the key insight in ZZ is the use of $f+1$ execution replicas in the normal case, e.g., there are no faults, and to activate additional sleeping replicas only upon failures. By multiplexing fewer replicas onto a given set of shared servers, ZZ can provide more server capability to each replica. In an application case of datacenter, e.g., multiple application targets hosting on a physical server, ZZ reduces the aggregate number of execution replicas running in the data center, improving throughput and response time. Besides, ZZ relies on the virtualization for fast replication activation upon failures, and enables newly activated replicas to immediately begin processing requests by fetching state on-demand. More clearly, ZZ is not a new "BFT protocol" that is typically used to refer to the agreement protocol; instead, ZZ is an execution approach that can be interfaced with existing BFT-SMAR agreement protocols. Based on that purpose, the prototype of ZZ is not to seek to optimize the agreement throughput, but to demonstrate the feasibility of ZZ's execution approach with a reasonable agreement protocol.

More precisely, ZZ was motivated by the observations that the request executions of BFT replication systems dominate the total cost of processing requests in BFT services, and the hardware capacity needed for request executions is far exceeded that for running the agreement protocol. The authors argue that the total cost of a BFT service can be practically reduced only when the total overhead of request executions, instead of the cost of reach a consensus, is somehow reduced. Another motivation is from the PBFT, which claimed "it is possible to reduce the number of copies of the state to $f+1$ but the details remain to be worked out". ZZ works on these motivations and distinguishes two request execution cases. In a normal case (or graceful case), ZZ enables general BFT services to be constructed with a replication cost close to $f+1$, halving the $2f+1$ or higher cost incurred by prior approaches, which in turn achieves higher throughput and lower response times for request executions. However, in the worst case (e.g., all applications experience simultaneous faults), ZZ requires an additional f replicas per application, matching the overhead of the $2f+1$ approach. In such cases that failures occur, ZZ incurs a higher latency to execute some requests until its failure recovery protocol is complete. The overall construction of ZZ still requires $3f+1$ agreement replicas. Also, the ability to quickly activate additional replicas upon fault detection is critical for ZZ since it is related to the response times of request executions.

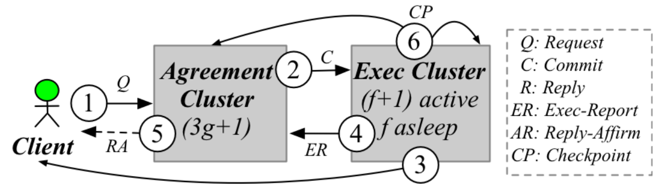


Fig. 28. The normal agreement and execution protocol in ZZ proceed through steps 1-5. Step 5 is needed only after a fault. Checkpoints (step 6) are created on a periodic basis [215].

The ZZ design distinguishes two types of replicas: agreement replicas that assign an order to a client request, and execution replicas that maintain application state and execution client requests. Replicas fail independently, and ZZ assumes an upper bound g on the number of faulty agreement replicas and a bound f on the number of faulty execution replicas in a given window of vulnerability (initially set to infinity), and an adversary may coordinate the actions of faulty nodes in an arbitrary manner. The safety of ZZ is guaranteed in an asynchronous network, and the liveness of ZZ is guaranteed only during periods of synchrony. From a high-level perspective on ZZ design, there exist two insights helping ZZ reducing the replication cost of BFT from $2f+1$ to nearly $f+1$. One is that if a system is designed to be correct in an asynchronous environment, it must be correct even if some replicas are out of date. For example, those replicas can be considered in a sleep mode and do not respond under the assumption of asynchrony. The other one is that, during fault-free periods, a system designed to be correct despite f Byzantine faults must be unaffected if up to f replicas are turned off. Based on the above insights, ZZ leverages the second insight on fault-free periods to turn off f replicas, which only require $f+1$ replicas to actively execute requests. However, when a failure occurs, ZZ leverages the first one on asynchrony assumption, and behaves exactly as if the f standby replicas were slow but correct replicas. The switch between two execution modes highly depends on a quick replica weak-up mechanism. Practically, virtualization can provide this capability by maintaining additional replicas in a "dormant" state. ZZ assumes that replicas are run inside virtual machines, and it is possible to run multiple replicas on a single physical server. By using virtualization, ZZ can fast replica activation, and optimizes the recovery protocol to allow newly activated replicas to immediately begin processing requests through an amortized state transfer strategy. Typically, ZZ consists of a six-step procedure, and Fig. 28 shows the normal agreement and execution protocol in ZZ. Besides, the authors implement a prototype of ZZ by enhancing the BASE library and combining it with the Xen virtual machine and the ZFS file system to evaluate the performance. For example, ZZ's use of only $f+1$ execution replicas in the fault-free case yields response time and throughput improvements of up to 66%. Interested readers can refer to Sections 4 and 5 of [215] for more details on these steps and implementations, respectively.

25) On-Demand Replica Consistency - (middleware): On-demand replica consistency is an extension to existing BFT architectures, aiming to increase performance for the default number of replicas, proposed by Distler and Kapitza in 2011 [216]. The proposed approach executes a request only within a selected subset of replicas, using a *selector* component co-located with each replica, to optimize the resource utilization of their execution states. The improved throughput under the common case is even beyond that of the corresponding unreplicated services. To avoid the divergent replica states, a selector on-demand updates outdated objects on its local replica prior to processing a request. Typically, an agreement-based BFT replication system consists of two stages: agreement and execution [176]. The agreement stage is responsible for imposing a total order on client requests, while the execution stage processes the requests on all replicas, preserving the order determined by the agreement stage to ensure consistency. Due to the requirements on consistency among all replicas, the maximum throughput achievable for a fault-tolerant service is bounded by the throughput of a single replica. The approach is motivated by the fact that to tolerant f faults, $f+1$ identical replicas provided by different replicas prove a reply correct. Thus, it can execute each request on only a subset of $f+1$ replicas during a common case operation. While, in case of failures, additional replicas are required to process the request. The subset of replicas to execute a request is selected for each request individually. In particular, the approach divides the service state into objects, and assigns each object to $f+1$ replicas.

In general, the states of replicas are not completely identical at all times but may differ across replicas, which depends on the requests a replica has executed. As clients may send requests accessing objects assigned to different replicas, a *selector* module co-located with every service replicas provides *on-demand replica consistency (ODRC)*. Also, the selector ensures that all objects possibly read or modified by a request are consistent with the current service state by the time the request is executed. However, other objects that do not belong to this domain are unaffected and may remain outdated. The approach can guarantee safety under an asynchronous model, and liveness under some bounded fair links. Also, the approach requires deterministic state machines. By separating agreement and execution stages, the agreement stage and execution stage can be located on different replicas. Besides replicas, the approach assumes that there exist voters, whose responsibility is to identify correct replicas, and the clients perform the functionalities of voters. To apply ODRC, a non-faulty replica should have the following properties: total request ordering, request execution, and reply cache. For total request ordering, it requires that even under an arbitrary sequence of client requests (e.g., may differ between replicas as inputs), the agreement stage outputs a stable totally-ordered sequence of requests, e.g., requiring identical across all replicas. For request execution, on the totally-ordered sequence of requests, each replica in the execution stage outputs a set of replies that is identical to the output of all other non-faulty replicas. For reply cache, each replica caches replies in order to provide them to voters on demand. To apply to ODRC, a non-faulty voter should have the following properties: reply verification and incomplete voting notification. For reply verification, as the voter receives $f+1$ identical replies from different replicas, the result of a client request should be accepted. For incomplete voting notification, all non-faulty replicas eventually learn about the incomplete voting.

In general, a selector module is deployed between the agreement stage and execution stage. Each replica has its own selector. Selectors of different replicas do not interact with each other, but rely on the same deterministic state machine and operate on the same input. Based on the totally-ordered sequence of requests from the agreement stage, all non-faulty selectors should behave in a consistent manner due to the deterministic state machine. The selectors ensure an ODRC consistency, and the ODRC consistency is “on demand" in two dimensions. One dimension is that the consistency is only ensured when a request to be executed actually demands it, called the time dimension; the other dimension is that the consistency is only ensured for the objects actually accessed by the request, called the space dimension. The proposed approach uses ODRC as a general term for applying selective request execution in conjunction with on-demand replica consistency. Besides, the ODRC scheme also requires a BFT system to provide a mechanism to checkpoint the service state and a mechanism to restore the service state based on the checkpoint. The evaluation shows that, with the proposed approach, the overall performance of a BFT NFS can be almost doubled, and even outperforms the unreplicated NFS system.

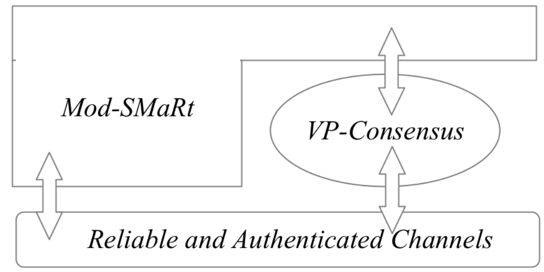


Fig. 29. Mod-SMaRt replica architecture. The reliable and authenticated channels layer guarantee the delivery of point-to-point messages, while the VP-Consensus module is used to establish agreement on messages to be delivered in a consensus instance [23].

26)MOD-SMART - (architecture): MOD-SMART is an acronym of Modular State Machine Replication, which is a transformation from Byzantine consensus to BFT state machine replication, proposed by Sousa and Bessani in 2012 [23]. In general, a solution to a consensus problem is at the core of any distributed SMR protocols, however, such protocols are monolithic, in which they do not separate clearly the consensus primitive from the remaining protocol. Actually, the consensus protocol can be considered as a black-box primitive, which can be further considered as a module in modular transformations. A latency-optimal protocol for BFT SMR, e.g., PBFT, typically requires at least three communication steps for its consensus plus two extra steps to receive the request from the client and send a reply back. MOD-SMART implements SMR using a special Byzantine consensus primitive called *Validated and Provable Consensus* (VP-Consensus), which can be easily obtained by modifying existing leader-driven consensus algorithms. MOD-SMART is a modular BFT SMR protocol built over a well-defined consensus module that requires only the optimal number of communication steps, i.e., the number of communication steps of a consensus plus two. In MOD-SMART, the VP-Consensus is a “grey-box" abstraction that allows the modular implementation of an SMR without using some reliable broadcast protocols. By doing so, it can avoid extra communication steps required to safely guarantee that all requests arrive at all correct replicas. And, the monolithic protocols can avoid those extra steps by merging a reliable broadcast with a consensus protocol. Also, MOD-SMART avoids mixing protocols by using some rich interfaces exported by VP-Consensus, which allows it to handle request timeouts and triggers internal consensus timeouts.

More specifically, MOD-SMART is a theoretical work, however, it can be implemented on some practical projects, e.g., BFT-SMART [217] (an open-source Java-based BFT SMR library). MOD-SMART as a modular BFT SMR protocol consists of three sub-phases: client operation, normal phase, and synchronization phase. Fig. 29 shows a general architecture of a replica. In general, MOD-SMART builds on top of a reliable and authenticated point-to-point communication substrate and a VP-Consensus implementation. Such modules may also use the same communication support to exchange messages among processes. MOD-SMART operates the VP-Consensus to execute a sequence of consensus instances, where

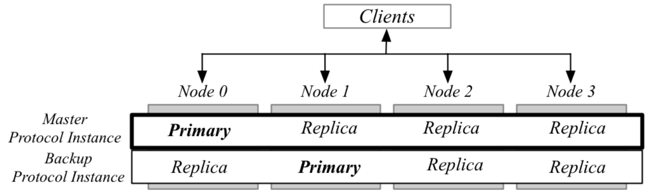


Fig. 30. RBFT overview ($f = 1$) [165].

each instance operates a batch of requests for its execution, and the same proposed batch is decided on each correct replica. During the normal phase, MOD-SMART assumes each correct replica can execute concurrently only on the current instance i and the previous consensus instance $i - 1$. All correct replicas remain available to participate in consensus instance $i - 1$ even if they are already executing i. When conditions of the normal phase are not satisfied, e.g., no agreed block generated, the synchronization phase may be triggered. By modulization an SMR BFT, it is easy to implement. Besides, the authors discuss how to make simple modifications to the leader-driven consensus algorithms to make them compatible with the transformation.

27) RBFT - (architecture): RBFT is a Redundant BFT protocol to execute multiple instances of a BFT protocol executed in parallel, proposed by Aublin et al. in 2013 [165]. These multiple instances are the same BFT protocol and each with a primary replica executing on a different node. This means that each instance has a primary replica, and various primary replicas are all executed on different machines. In general, it requires all instances to order the requests of clients, but only one instance, namely *master instance*, effectively executes them. And all other instances, namely *backup instances*, order requests in order to compare the throughput they achieve with that achieved by the master instance. In case that a master instance is slower than its backup instances, the primary of that master instance is considered to be malicious and replicas elect a new primary, at each protocol instance. Also, RBFT implements a fairness mechanism between clients by monitoring the latency of requests, which assures that client requests are fairly processed.

In general, the leader-based BFT protocols are not robust because they rely on a dedicated replica, the primary, to order requests. Even if some mechanisms in literature have been exploited to detect and recover from a malicious primary, it still has a chance that a primary smart enough to launch attacks. Based on the analysis on prior "claimed" robust BFT protocols, e.g., Prime, Spinning, and Aardvark, these protocols are not effectively robust enough. Most of these robustness are based on some strong assumptions, and a primary node can be smartly malicious and causes huge performance degradation without being detected. For instance, Prime is robust under a certain level of network synchrony, and if the variance of the network is too high, there exists a chance that a malicious primary can heavily affect the performance of the system. Similarly, Aardvark is based on the static load, and Spinning is based on the correct primary. For example, the malicious primary in Spinning can delay the sending of the ordering messages up to maximal allowed time, which is hard to be identified as a faulty replica. From a high-level perspective, similarly to these robust protocols, RBFT utilizes replicas to monitor the throughput of the primary and trigger the recovery mechanism when the primary is low.

In more details, RBFT requires $3f + 1$ replicas (i.e., $3f + 1$ physical machines) to tolerate f failures. Each replica runs $f + 1$ protocol instances of a BFT protocol in parallel, as shown in Fig. 30. Theoretically, $f + 1$ protocol instances are sufficient to detect a faulty primary and ensure the robustness of the protocol. This indicates that each of the N replicas in a system can run locally one replica for each protocol instance, in which different instances order the requests following a PBFT like protocol. As shown in Fig. 30, primary replicas of various instances are distributed on replicas and the distribution should follow certain rules: at any time, there is at most one primary replica per node. All instances participate in the order of client requests, but only the requests ordered by the master instance are executed by replicas, and the main responsibility of backup instances is to monitor the master instance. Each replica runs a monitoring module that computes the throughput of the $f + 1$ protocol instances. If $2f + 1$ replicas observe that the ratio between the performance of the master instance and the backup instance is lower than a given threshold, then the primary of the master instance is considered to be malicious, and a new one is required to elect. To increase the overall throughout, RBFT can leverage multicore architecture to run multiple instances of the same protocol in parallel.

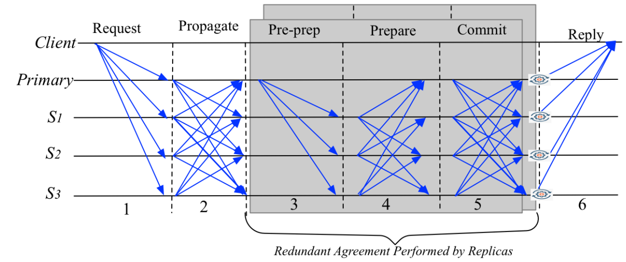


Fig. 31. RBFT protocol steps ($f = 1$) [165].

Fig. 31 shows an abstract message pattern of RBFT protocol in the case of $f = 1$, which is a six-step protocol. Simply, in step one, the client sends a request to all nodes; in step two, the correct node propagates the request to all nodes; in step three, four, and five, the replicas of each protocol instance execute a three-phase commit protocol to order the request; and in the last step, nodes execute the request and send a reply message to the client. For more detailed information on each step, interested readers can refer to Section IV of [165].

28)ClusterBFT - (middleware): ClusterBFT exploits the opportunities to integrate BFT into cloud applications, proposed by Stephen and Eugster in 2013 [218]. By leveraging a BFT replication system, it provides a conceptual design and implementation to the existing cloud-based data analysis to provide integrity, availability, and confidentiality of cloud services. In typical cloud data-flow processing applications, the generated data-sets (e.g., from analysis and correlation with existing data-sets) must be trustworthy. While prior existing cloud services, the lack of trust on various facets are popular, and BFT protocols can help to establish this trust without compromising integrity and availability. More specifically, ClusterBFT, for cloud-based assured data processing and analysis, creates sub-graphs from acyclic data-flow graphs that are then replicated. It performs a BFT consensus at less overhead system-level, rather than at client request level, which enables the system to dynamically adapt to changes in required responsiveness and perceived threat level as well as to dynamic deployment. ClusterBFT combines variable-grain clustering with approximated and offline comparisons, separation of duty, and smart deployment to keep the overhead of BFT replication low while providing good fault isolation properties. ClusterBFT considers a

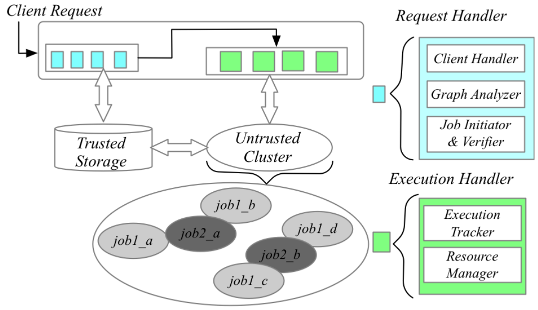


Fig. 32. Architecture of ClusterBFT [218].

system that is deployed on a cloud service that leases out virtual machine (VM) to users, and each VM as a node. It distinguishes two adversary models: a strong adversary, which can manipulate all internal aspects of a node and collude with other malicious nodes, and a weak adversary, which shares the same properties of a strong adversary (but may only cause omission or commission faults).

In general, adopting a BFT replication scheme to a cloud provides several intuitive benefits, such as attribution, portability and interoperability, determinism, and heterogeneity. However, this adoption indeed has some challenges, e.g., scalability, granularity, and rigidity. To address these challenges, ClusterBFT follows some design principles, e.g., variable granularity, variable, replication, approximate, offline redundancy, separation of duty, and fault isolation. Interested readers on these detailed challenges and design principles can refer to Section 3 of [218]. Following these design principles, ClusterBFT has several components, as shown in Fig. 32, e.g., request handler, execution handler, and fault identification and isolation. The request handler component is the control tier, where it accepts scripts submitted by the client and submits the script for execution. This handler further consists of three logical sub-components: client handler, graph analyzer, and job initiator and verifier. Within the execution handler, it consists of two sub-components: execution tracker and resource manager. The main functionalities of fault identification and isolation are used to identify faults with the help of a fault analyzer (e.g., not collecting enough $f + 1$ digests from output verifier) and to isolate the identified faults.

29)BFT Selection - (middleware): BFT Selection is a mechanism to help users to choose the most convenient protocol according to the preferences of BFT user, proposed by Shoker and Bahsoun in 2013 [219]. Many BFT protocols have been introduced in replication systems, with their advantages and disadvantages. Typically, it is a difficult task to select a right or optimal BFT algorithm for an application. However, the service of an application can be quantified mathematically. Based on this observation, the selection algorithm applies some mathematical formulas to make the selection process easy and automatic. The selection algorithm selects a 'preferred' BFT protocol, the one that matches user preference the most, among a set of candidates. The selection algorithm has an evaluation process in charge of matching the user preference according to both reliability and performance. And the selected protocol typically has the highest evaluation mathematical score. Two types of indicators are adopted: Key Characteristic Indicators (KCI) and Key Performance Indicators (KPI). Specially, the KCI are the properties (with boolean values) of a protocol, which can strictly decide whether an evaluated protocol could be selected or not, and the KPIs are the properties that evaluate the performance of the protocol like throughput, and latency.

The selection mode can distinguish three different categories: static, dynamic, and heuristic. The static mode is the one that the user chooses a protocol only once, and this mode can only be changed when the service is rebooted. A possible application for this mode is to use it in clouds. The dynamic mode makes the system react dynamically to the changes of the system state, which allows the user to run multiple protocols, where a running protocol can be stopped and another protocol is launched after performing the selection process. Typically, this model fits the applications that as the underlying system state changes, the performance of protocols differs a lot. The heuristic model is much similar to the dynamic mode, the ability to allow the user to modify the weights (e.g., preferences) as the system state changes using some predefined heuristics. Much of this work focuses on the mathematical perspective.

30) BFTRaft - (feature): BFTRaft is a Byzantine variant of the Raft consensus protocol, inspired by the original Raft [55] and PBFT algorithm, proposed by both Copeland and Zhong in 2014 [220]. From a high-level perspective, BFTRaft maintains the properties on safety, fault tolerance, and liveness of the Raft in the presence of Byzantine faults with the goals of simplicity and understandability. Like Paxos, Raft targets a crashed fault model. When applying the Byzantine fault model directly to Raft, the safety and availability of Raft would be compromised in several aspects, e.g., leader election and log replication. More specifically, any node in the original Raft can trigger an election and terminate the current term, thus a Byzantine node can effortlessly starve the whole system by perpetual elections, and subvert Raft's availability. In Raft, the leader typically serves as a single point of contact between the client and the rest of the system, this would leave a great chance for a Byzantine leader to perform some malicious behaviors, e.g, modify client's request, which may cause the safety issues.

BFTRaft adopts similar technologies and decomposition to preserve the simplicity and understandability of Raft with several modifications and additions to provide Byzantine fault tolerance. More precisely, BFTRaft enables the following aspects: *message signatures* to authenticate messages and verify their integrity, *client intervention* to allow clients to interrupt the current leadership if no progress, *incremental hashing* to a compute a blockchain similar cryptographic hash when appending a new entry to its log, *election verification* to let a quorum of nodes to verify the current leader, *commit verification* to broadcast messages to each other (like PBFT) rather than just to the leader, and *lazy voters* to delay the vote to a candidate unless it believes the current leader is faulty. Similar to Raft decomposition, the BFTRaft algorithm also decomposes the consensus problem into two relatively independent sub-problems: log replication and leader election. By enabling the above features, the BFTRaft offers the same safety guarantees as Raft.

BFTRaft uses $3f + 1$ replicas to tolerate up to f Byzantine failures. In BFTRaft, each node is in one of the three roles: leader, follower, or candidate. During the leader election, a winner of the election serves as the leader for the rest of the term (a time unit of a consensus round). In general, BFTRaft can maintain a high level of coherency between nodes' view of the current term. The authors also implement a proof-of-concept of BFTRaft in Haskell, and its code is available on GitHub (https://github.com/chrisnc/tangaroa). Interested readers can refer to their work for more details [220].

31) BChain - (architecture): BChain is a chain-based BFT protocol, where replicas are organized in a chain, proposed by Duan et al. in 2014 [221]. In general, chain-based protocols aim to achieve

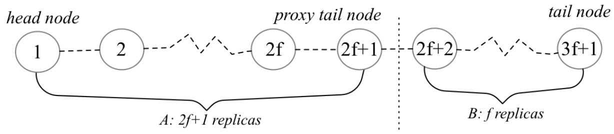


Fig. 33. BChain-3 Replicas are Organized in a China [221].

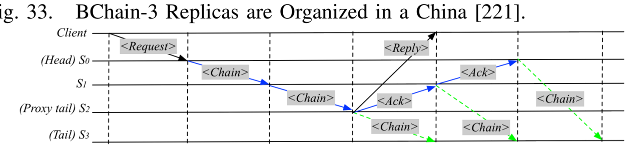
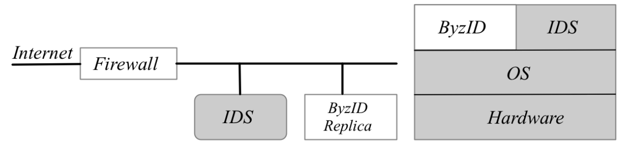


Fig. 34. BChain-3 common case communication pattern. No conventional broadcast is used at any point [221].

high throughput at the expense of higher latency, while BChain can achieve a comparably performance to other protocols in fault-free cases, and when failures occur, it can also quickly recover to its steady-state performance. The key idea behind BChain is a Byzantine failure detection mechanism, called *re-chaining*, where faulty replicas are placed at the end of the chain, until they can be replaced. From a high-level perspective, replicas in BChain are organized in a chain structure. During normal case executions (best case scenarios), clients send their requests to the *head* of the chain, which orders the requests. Then, the ordered requests are forwarded along the chain and executed by replicas. Once a request reaches a replica called *proxy tail*, a reply will be sent back to the client. However, when a failure occurs, BChain employs re-chaining to reorder the chain, and the head performs this task. The main goal of re-chaining is to make sure that a fault cannot affect the critical path. Each replica is monitored by its successor along the chain, upon detecting a suspicion, the head issues a new chain ordering where the accused replicas are moved out of the critical path, and the accuser is moved to a position in which it cannot continue to accuse others. The re-chaining approach is inexpensive; and a single re-chaining request can proceed as a single client request.

Replica in BChain is organized in a metaphorical chain, as shown in Fig. 33. Each replica is uniquely identified by its identity, e.g., $\{p_1, p_1, ..., p_n\}$. Initially, the replicas' IDs are numbered in ascending order, which is subject to change during the re-chaining process. The first replica in the chain structure is called the *head*, denoted p_h, the last replica is called the *tail*, and the $(2f+1)^{th}$ replica is called the *proxy tail*, denoted p_p. There are two distinct subsets along the chain. The first subset A contains the first $2f+1$ replicas, initially p_1 to p_{2f+1}, and the second subset B contains the last f replicas in the chain, initially p_{2f+2} to p_{3f+1}. In general, the chain order is maintained by every replica and can be changed by the head, and is communicated to replicas through message transmission.

In general, BChain works under an asynchronous environment. Like many other protocols, its safety holds in an asynchrony model, and its liveness is ensured by assuming the existance of partial synchrony. BChain comes two variants, BChin-3 and BChain-5. Under the same setting, e.g., tolerating f failures, BChain-3 requires $3f+1$ replicas and a reconfiguration mechanism coupled with the detection and re-chaining algorithms, while BChain-5 requires $5f+1$ replicas but can operate without the reconfiguration mechanism. More specifically, BChain-3 has five sub-protocols: chaining, re-chaining, view change, checkpoint, and reconfiguration; while BChain-5 is similar to BChain-3 except without a reconfiguration protocol. Both BChain variants have an identical message flow. Briefly, the chaining protocol orders clients' requests, while the re-chaining process recognizes the chain in response to failure suspicions. Faulty replicas are moved to the end of the chain. The view change protocol selects a new head when the current head is faulty, or the system is slow. The checkpoint protocol is similar to that of PBFT, which is mainly used to reduce the cost. The reconfiguration protocol is responsible for reconfiguring faulty replicas.



(a) The IDS Interface at a Replica (b) IDS Implementation

Fig. 35. The IDS/ByzID architecture. Gray components is trusted [222].

Fig. 34 shows the BChain-3 common case message communication pattern. BChain contains two types of messages along the chain: $< CHAIN >$ messages transmitted from the head to the proxy tail, and $< ACK >$ messages transmitted in reverse from the proxy tail to the head. A request is *executed* after a replica accepts the $< CHAIN >$ message, and a request *commits* at a replica if it accepts the $< ACK >$ message. Interested readers can refer to the work on both BChain-3 and Bchain-5. Besides, there also exist some similar works on chain-based protocols, e.g., Aliph-Chain [82].

32) ByzID - (feature): ByzID is a primary-based Byzantine replication protocol by combining an Intrusion Detection System (IDS), provided by Duan et al. in 2014 [222]. By integrating the IDS via Byzantine failure detector, ByzID itself is considered as a BFT protocol that has cost comparable to crash-resilient protocols like Paxos. Practically, replicated state machines (RSM) and IDS are distinct protocols to provide availability and integrity of critical network services. An IDS is a tool for near real-time monitoring of host and network devices to detect events that could indicate an ongoing attack. It has three major types: anomaly-based detection [223], misuse-based detection [224], and specification-based detection [225]. However, all three types are not perfect, and an IDS itself not only suffers from deficiencies that limit its utility (e.g., false positives induced by the human administrator and false negatives when an ongoing attack is not detected), but also is not resilient to crashes. ByzID provides a unified approach to improve RSM resilience by leveraging intrusion detection, rather than using each technique independently. It utilizes a lightweight specification-based IDS as a failure detection component to build a Byzantine-resilient RSM. In specification-based IDS, any sequence of operations outside of the specification is considered to be a violation. By integrating IDS, ByzID provides two main advantages: 1) its efficiency is comparable to its crash failure counterpart, and 2) its robustness protects against a wide range of failures (e.g., flooding, timing, and fairness attacks). But, ByzID cannot protect against all possible attacks, only those that IDS can help with, as IDS only captures the network packets of the protocol and analyzes them according to the specification. Fig. 35 shows an IDS/ByzID architecture.

More specifically, ByzID is a primary-based RSM protocol, in which a primary receives client requests and coordinates the other replicas (as backups). In the event of replica failure, a new replica runs a configuration protocol to replace the failed one. Typically, the primary reconfiguration runs *in-band*, and other replicas must wait until the primary reconfiguration competes. However, the reconfiguration for other replicas can run *out-of-band*, where replicas continue to run the protocol without stopping for reconfiguration. ByzID relies on monitoring instead of ordering to detect failures,

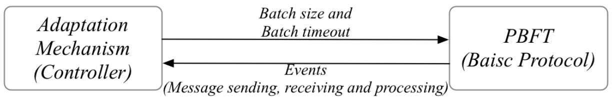


Fig. 36. aPBFT control loop [227].

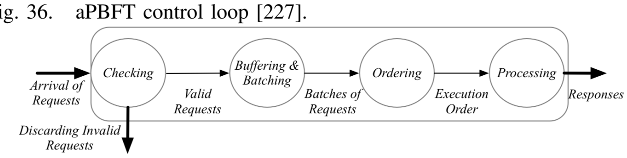


Fig. 37. Pipleline: an abstract for request processing in aPBFT [227].

e.g., using a *trusted* specification-based IDS to detect and suppress the primary equivocation, enforce fairness, etc. It uses the IDS to monitor the behavior of replicas, whose implementation can be in a form of a small state machine (e.g., via BroIDS framework [226]). Each replica is associated with a separated IDS component. Even when an IDS experiences a crash, its hosting replica can still continue to process requests. Besides, the ByzID can be deployed in a LAN network with a simple rooted-tree structure, where the primary is the root and the backups are its direct siblings (leafs). In general, BzyID protocol uses $2f+1$ replicas to tolerate f failures, and it only require *three* communication rounds to reply a client's request. The safety of ByzID holds in any asynchronous environment, and its liveness can be ensured under the assumption of a partial synchrony.

ByzID highly depends on the component of a Byzantine failure detector. To make the primary order message correctly, it requires some IDS specifications for Byzantine failure detectors. Specially, a Byzantine failure detector has four specifications to follow: consistency (preventing the primary from sending "inconsistent" order messages to other replicas), no gap (preventing the primary from introducing gaps in a message ordering), fairness (ensuring RSM to execute the client requests in a FIFO order), and timely action (used for detecting a crash-stop and slow primary). Interested readers can refer to the work [222] for more details.

33) Adaptive PBFT - (feature): Adaptive PBFT (aPBFT) is an extension to the original PBFT protocol with a feature of dynamic configuration of request batching parameters, proposed by de Sá et al. in 2013 [227]. The request batching mechanism is one of the techniques to optimize the PBFT-based protocols, and most of these protocols are based on some static configurations on a target distributed system. However, when the target distributed system is dynamic, e.g., the underlying characteristics change dynamically (i.e., workload, channel QoS, network topology), the configuration of a request batching mechanism must follow the dynamics of the system or it may not yield the desired performance improvement. For instance, if the prescribed batch timeout is too large for rather inactive clients, it may incur a worse performance for the use of request batching optimization compared with the non-optimized version. Adaptive PBFT is an approach for automatic tuning of PBFT parameters, based on a feedback control loop, which can constantly sensor both the replication protocol performance and the underlying distributed system behavior. The proposed adaptive mechanism focuses on the parameters-related batching mechanism and specifically for the PBFT family protocols. It has the ability to self-tuning the batching mechanism at run-time.

In more details, adaptive PBFT is an adaptive extension of PBFT, which promotes a dynamic adaptation of the batching mechanism, namely, the batch size (BS) and the batching time (BT). To dynamically adjust the scheme according to the application workload patterns, adaptive PBFT continually estimates client workloads and the performance of the basic PBFT protocol, which can be achieved by implementing a feedback control loop. This feedback control needs to collect some PBFT information, e.g., on the transmission, reception, and processing of messages, to adjust the parameters of the batching mechanism. Fig. 36 shows an abstract model of on adaptive PBFT control loop, whose operations can be pipelined, as shown in Fig. 37. Simply, a pipeline adaptive control loop consists of four stages: checking (verify the integrity, authenticity, and validity of the client requests), buffering and batching (valid requests kept in the buffer for batching), ordering (run an agreement on the request order), and processing (process requests and send related responses to the requesting clients). Interested readers can refer to the work [227] for more details.

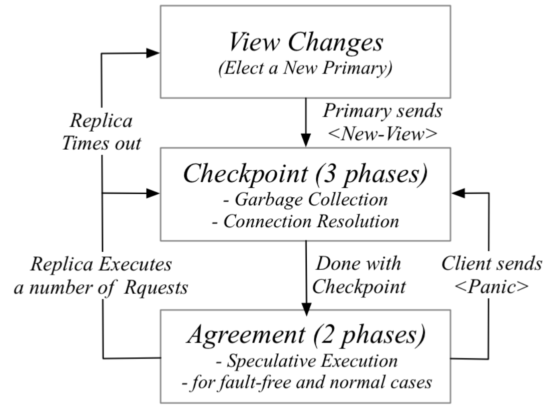


Fig. 38. Layered structure of hBFT [228].

34) hBFT - (feature): hBFT is a leader-based speculative Byzantine fault-tolerant protocol with an optimal resilience, proposed by Duan in 2014 [228]. hBFT uses a speculation to reduce the cost of Byzantine agreement, while also maintaining an optimal resilience, utilizing $3f+1$ replicas to tolerate f failures. By utilizing a speculation, correct replicas may be temporarily inconsistent, and hBFT employs a three-phase PBFT-like checkpoint sub-protocol for both garbage collection and contention resolution. Two events can trigger the checkpoint sub-protocol, either by the replicas when they execute a certain number of operations or by clients when they detect the divergence of replies. From a high-level perspective, hBFT offers a better performance by moving some critical jobs to the clients while minimizing side effects that can actually reduce performance. Also, there is no additional cost by moving some critical jobs to the clients, with the benefits of simplifying the design and reducing message complexity. Thus, this frees the replicas to run expensive protocols to establish the order for every request. Also, hBFT can tolerate an unlimited number of faulty clients. Besides, hBFT has the same operations for both fault-free and normal cases.

Fig. 38 shows an abstract on layered structure of hBFT. In more details, it includes four main components: agreement, checkpoint, view change, and client suspicion. hBFT employs the same agreement protocol for both fault-free and normal cases, and utilizes a three-phase checkpoint sub-protocol for contention resolution and garbage collection. The checkpoint sub-protocol can be triggered by replicas when they execute a certain number of requests by clients if they detect divergence of replies. The view change sub-protocol guarantees the liveness and can coordinate the change of the primary, and it can occur during normal operations or in the checkpoint sub-protocol.

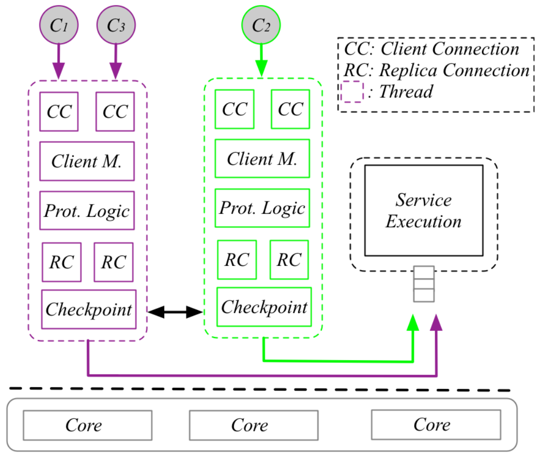


Fig. 39. Replica with self-contained pillars of a Consensus-oriented Parallelization (COP) [229].

And, the client suspicion sub-protocol prevents faulty clients from attacking the system.

35) Consensus-oriented Parallelization - (architecture): Consensus-oriented parallelization (COP) is a scheme to execute the consensus efficiently, which disentangles consecutive consensus instances and executes them in parallel by independent pipelines, proposed by Behl in 2015 [229]. In general, this work requires some background on system parallelization to understand the key ideas behind the proposed COP scheme. The authors extensively discussed the disadvantages of a traditional approach to executing instances in a pipelined fashion and the need for parallelization at a consensus level. The COP's basic approach is to lower the dependencies between consecutive consensus protocol instances, to process them in independent pipelines, and to parallelize these pipelines in their entirety and not the single stages they are composed of. Briefly, one can think of COP as a superscalar design, where the throughput of consensus instances scales well with the number of operations a system has to process concurrently as long as there are enough computing and network resources available. In more details, the authors try to identify important tasks a replica needs to perform over and over again to provide its services, e.g., message authentication, agreement, execution, and checkpointing. Also, the authors discuss the disadvantages of task-oriented parallelization, e.g., limited throughput, limited parallelism, asymmetric load, and high synchronization overhead, etc. The authors proposed to structure replicas around the instances of the consensus protocol employed within the system to agree on operations.

Fig. 39 shows a conceptual framework on consensus-oriented parallelization. Each consensus instance is assigned to and executed by a container, called *pillar*, and each pillar runs in a dedicated thread, processing private copies of all functional modules required to perform the entire consensus protocol and client handling. We can literally consider each pillar runs under a separated environment. Pillars do not intend to share any modifiable state as far as possible, and they reply private connections to other replicas. Pillars work asynchronously and, if communication is needed with other components of a replica, they exclusively use an in-memory message passing. For example, if a pillar running at the leading replica receives a request from a client or requests to create a batch, it imitates a new consensus instance by proposing it to the other replicas via its connections to them. As each pillar runs in parallel and cannot provide a total order on its own, the execution stage has to enforce the order by means of the instances' sequence numbers before it invokes the service implementation. Via COP, it can provide some advantages to current consensus designs, e.g., scaling throughput and parallelism, symmetric load, reduced synchronization overhead, and conciliated decisions. Interested readers can refer to the original work [229].

36) Elastico - (blockchain): Elastico is a distributed agreement protocol for permissionless blockchains, proposed by Luu et al. in 2016 [230]. By using sharding technology, Elastico can scale transaction rates almost linearly with available computation for mining. For example, the more computation power in the network, the higher the number of transaction blocks selected per unit time. The number of committees grows proportionally to the total computation power in the network. All committees, each of which has a small constant number c of members, run a classical BFT consensus protocol internally to agree on a block. Elastico can tolerate Byzantine adversaries up to one-fourth of the total computational power. Essentially, Elastico uniformly partitions or parallelizes the mining network into smaller committees, each of which processes a disjoint set of transactions, or "shards". The technology of sharding is common in a non-Byzantine environment, e.g., databases, and Elastico is used for a secure sharding protocol in the presence of Byzantine adversaries. A permissionless sharding protocol typically consists of five critical components in each consensus round [1]: identity establishment and committee formation, overlay setup for committees, intra-committee consensus, cross-shard transaction processing, and epoch reconfiguration. The identity establishment and committee formation are used to establish an identity for each node before joining the protocol, and each identified node is then assigned to one committee. This process needs to prevent the Sybil identity [231], since Elastico targets permissionless blockchain. The overlay setup for committees is used to discover its neighboring nodes within the committee, and this process can be done with a gossip protocol [232]. Elastico assumes the overlay of a committee is a fully connected subgraph containing all the committee members. The intra-committee consensus runs a standard consensus protocol to agree on a single set of transactions. The cross-shard transaction processing is used to handle the transactions that are involved in more than one shards. And this process typically requires a kind of "relay" transaction to synchronize among related shards. The epoch reconfiguration is used to reconfigure the shards to guarantee the security of the overall system. This process typically requires some kind of randomness to prevent sophisticated attacks. The above five components are the most critical ones for a permissionless blockchain sharding. It also can be adapted to a permissioned blockchain setting.

In more details, Elastico performs under a static, round-adaptive adversary, and processors controlled by the Byzantine adversary can be arbitrarily malicious. And the round-adaptive adversary can select which processors to corrupt at the start of each run which means once the protocol begins its run, the choices of compromised processors are fixed. Technically, Elastico is a probability-based consensus, and it can guarantee the consensus protocol outputs the correct results with a high probability. In each consensus epoch, each participant solves a PoW puzzle based on epoch randomness obtained from the last state of the blockchain. Typically, within a shard, the shard members run a Byzantine fault-tolerant protocol, e.g., PBFT in Elastico, to get an agreement within its shard.

Even via the sharding scheme, Elastico improves the performance on throughput and latency for blockchain scenarios, however, it still has some aspect to improve [42]. 1) Elastico requires all participants

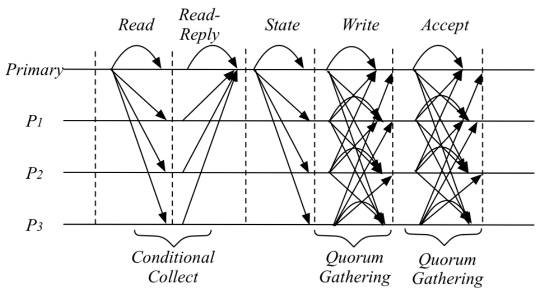


Fig. 40. VFT-SMaRt communication pattern [234].

to re-establish their identities, e.g., via solve PoWs, and re-build all committees in every epoch, and this would definitely involve a large communication overhead. Further, this will incur a significant latency that scales linearly with the network size as the protocol requires more time to solve enough PoWs to fill up all committees. 2) In its intra-committee consensus, Elastico requires a small committee size (e.g., about 100 participants) to limit the overhead of running PBFT in each committee. However, a small committee may increase the failure probability of the overall protocol. 3) During the last stage, the randomness used for establishing identity and committees can be biased by an adversary, the generated randomness is not a "good" randomness (e.g., with the feature of unbiased, unpredicted) [233], which allows malicious nodes to pre-compute PoW puzzles. 4) Elastico requires a trusted setup for generating initial common randomness that is revealed to all parties at the same time. 5) Elastico only tolerates up to a $1/4$ fraction of faulty participants even with a high failure probability, and its liveness needs to improve.

37) VFT - (architecture): VFT is an acronym of Visigoth Fault-Tolerant protocol which essentially is a reliable stateful service, proposed by Porto et al. in 2016 [234]. VFT introduces the Visigoth model which makes it possible to calibrate the timing assumptions of a system using a threshold of slow processes or messages, and also to distinguish between non-malicious arbitrary faults and correlated attack scenarios. The main observation behind of Visigoth model is that the Byzantine model is too pessimistic, especially for some relatively secure environment such as data centers. The Byzantine adversary's strength forces protocols to incur an unnecessary cost of $3f + 1$ replicas. BFT is designed to cope with coordinated malice, which is unlikely to happen within the security perimeter of a data center. This is based on the observation that data centers are more predictable and controllable than an open Internet environment, in order to make stateful services more resource-efficient. More often, data centers experience data corruption faults (e.g., bit flips) and these also represent a type of arbitrary behavior. To handle these requirements to tolerate different types of arbitrary faults, VFT observes that it is very unlikely for data corruption to affect the same data across multiple replicas. It is important to distinguish the correlation of these types of faults, e.g., the number of each type.

In more details, VFT proposes a customizable model that defines a limit of u faults, that bounds the number of allowed omission (i.e., crash) and commission (i.e., arbitrary) faults, a limit of o correlated commission faults, a limit of r arbitrary faults and, for every process p_i, s correct processes p_j that are slow with respect to it [235]. VFT can reduce the replication factor to solve a fundamental consensus problem from $n \geq 2u + r + 1$ (or $n \geq 3f + 1$ in the traditional formulation) to $n \geq u + s + o + 1$ (or $n \geq f + s + o + 1$) compared with an asynchronous BFT system. This benefit comes at a cost

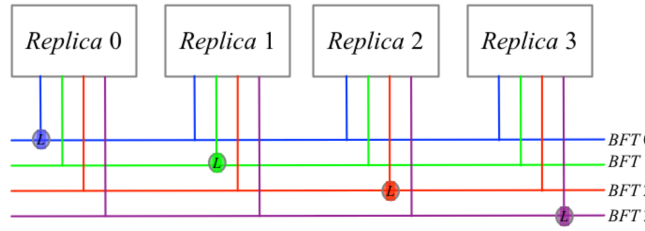


Fig. 41. In the fault-free case, each replica leads one of the BFT agreement protocol instances [236].

of making an additional assumption, whose overall correctness is at stake when these assumptions are not met. By parameterization, the system allows an administrator to configure the system to tolerate any combination of faults. The authors also propose a VFT state machine replication protocol, which is adapted from the BFT-SMART, called VFT-SMART. The message communication pattern of VFT-SMART, as shown in Fig. 40, is identical to that of BFT-SMART with two core message patterns. One pattern is that the first two message steps are executed during epoch changes to collect information about previous epochs and disseminate this information to replicas; the other pattern is that the final sequence of two all-to-all communication steps after the leader relays the client request, which drives all 'common case" execution. The authors also designed some few primitives, e.g., the Quorum Gathering Primitive (QGP), which forms the basis for the all to all communication pattern of the normal case operation. With this primitive, the gather tries to collect a larger quorum of $N - s$ replies, which can be seen as collecting replies from the fast but potentially faulty replicas. If a timeout occurs, the quorum is reduced to $N - u$, and this can be seen as collecting replies from correct but possible slow replicas. In general, this reduced quorum size is enough to ensure an intersection since x replicas did not reply by the first timeout and only s may be slow, $x - s$ replicas must have crashed and will not participate in future quorums.

The authors implemented a VFT protocol for a state machine replication library, and ran several benchmarks, the evaluation shows that VFT can tolerate uncorrelated arbitrary faults with resource efficiency and performance that resembles that of CFT systems instead of the BFTs.

38)SAREK - (architecture): SAREK is a parallel ordering framework that instantiates multiple single-leader-based BFT protocols independently, proposed by Li et al. in 2016 [236]. The framework partitions the service state to exploit parallelism during both agreements as well as execution, by utilizing request dependency which is abstracted from application-specific knowledge for a service state partitioning. One of the goals of SAREK is to distribute the extra workload bound over all replicas and enable parallelism at both the agreement and the execution stages. This is based on the observation that most existing single-leader BFT systems are typically far too pessimistic. For example, in a key-value store and many web applications, only a small fraction of requests interfere with each other. Thus, instead of having one leader at a time for the entire system, it uses one leader per partition and only establishes an order on requests accessing the same partition. Meanwhile, it supports operations that span multiple partitions and provides a deterministic mechanism to atomically process them. SAREK provides parallelism at both the agreement stage where requests are ordered, and the execution stage where they are processed. By enabling concurrent request executions, SAREK is required to exploit parallelism during the request ordering.

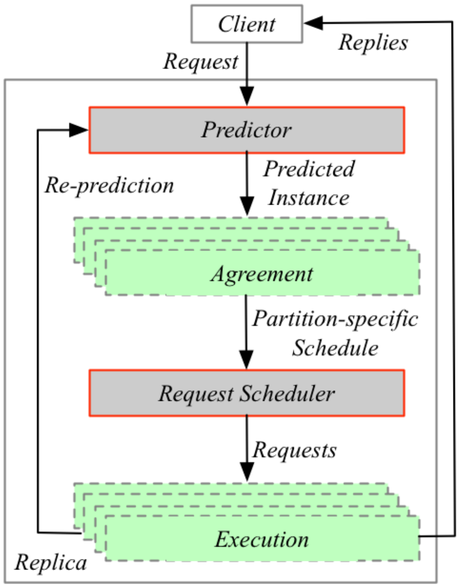


Fig. 42. System architecture of SAREK [236].

More specifically, SAREK partitions the service state and only linearly orders the requests accessing the same partitions, by creating a partition-specific schedule, so that independent requests on the agreement can be handled concurrently. Each partition selects one leader, which can help the overall system to balance the load-induced across all replicas. Once the agreement in a partition is complete, a dedicated execution instance is responsible for processing requests in each partition according to a local schedule. However, there exists a case where requests access multiple partitions, called "cross-border requests", and some special requirements should be met. For example, a cross-border request must not be executed more than once, and the processing of that request must be consistent with individual schedules that are determined for all affected partitions. To handle a cross-border request, SAREK uses a mechanism that is based on a combination of prioritizing partitions and safe reordering of requests. For example, only the execution instance of the partition with the highest priority actually processes a cross-border request while the instances of other involved partitions are put on hold in the meantime. Besides, to schedule the requests among the partitions, SAREK relies on some application-specific knowledge to define service-state partitions and to predict which partition a request should operate on. In general, the application-specific knowledge is essential to determine to which partition a state object belongs to. And each replica holds a deterministic *PREDICT()* function, which identifies the state objects to be read or written during the processing of a particular request.

SAREK uses $3f + 1$ replicas to tolerate up to f Byzantine failures. From a high-level perspective, SAREK can be based on an existing BFT implementation without requiring modifications to the most complex part, e.g., the agreement protocol. Instead, the agreement stage can be considered as a black box that is instantiated multiple times, once for each partition. Fig. 41 shows a fault-free case of SAREK, where a replica hosts multiple BFT instances and each instance manages a partition of the service state. Fig. 42 shows the system architecture of the SAREK framework. The agreement is the same as the single-leader BFT protocol in order to make SAREK compatible with most common BFT protocols that feature a separation of the agreement and the execution stages. To handle

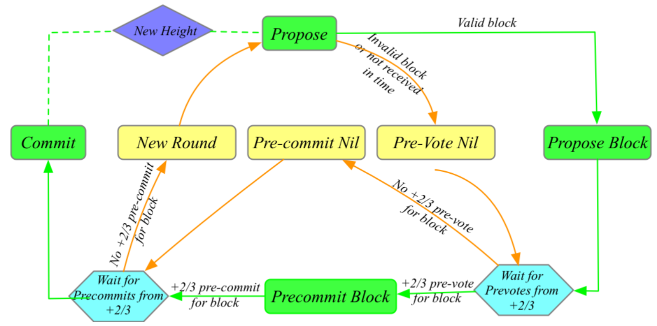


Fig. 43. After the proposal step, validators only make progress after hearing from two-thirds or more (+2/3) of other validators. The dotted arrow extends the consensus into atomic broadcast by moving to the next height [237].

a cross-border request, additional actions must be assured, e.g., the request ordered by all predicted BFT instances and requests executed exactly once. The authors provided a detailed description of the cross-border request agreements and the cross-border request execution. Interested readers can refer to Section IV for more details.

39) Tendermint - (blockchain): Tendermint is a secure state-machine replication algorithm in a blockchain paradigm, by providing a form of BFT-ABC (Atomic Broadcast) to offer accountability, proposed by Buchman et al. in 2016 [237], and its first development by Kwon in 2014 [238]. It is the first showcase on how the BFT consensus can be achieved in blockchain systems [38]. Tendermint consists of two major components, a consensus engine known as Tendermint core and its underlying application interface called the Application Blockchain Interface (ABCI). The Tendermint core is responsible for deploying the consensus algorithm, while the ABCI can be used to deploy any blockchain applications using any programming language. Tendermint begins with a set of validators, identified by their public keys, for proposing new blocks and voting on them. Each block is assigned with an incrementing index, or height. At each height, validators take turns proposing new blocks in rounds, so that for any given round, at most one valid proposer exists. However, due to the asynchrony of the network, it may take multiple rounds to commit a block at a given height. Tendermint uses $3f + 1$ replicas to tolerate up to f Byzantine failures. In general, validators engage in a two-phase voting process on a proposed block before that block is committed, by following a simple locking mechanism, it can prevent any malicious coalition of less than one-third of the validators from compromising the safety. In each round, the consensus algorithm contains three main components: *proposals, votes*, and *locks*.

In more details, Tendermint is based on a partial synchrony. In *proposals*, a new block must be proposed by a correct proposer at each round, then gossiped to other validators. If a proposal is not received in sufficient time, that proposal should be skipped. In *votes*, a two-phase voting scheme is adopted: *pre-vote* and *pre-commit*. A set of pre-commits from more than two-thirds of the validators for the same block at the same round is a valid commit block. In *locks*, Tendermint ensures that no two validators commit a different block at the same height. This is achieved by using a locking scheme that determines how a validator may pre-vote or pre-commit depends on previous operations at the same height. Fig. 43 shows a state transition for each validator. At the beginning of each round, a new proposer proposes a block, which needs a two-stage voting mechanism before it is committed to the blockchain. So a normal case would go through the following patterns: Propose $\rightarrow$ Propose Block $\rightarrow$ "Wait for pre-vote from $+2/3$" $\rightarrow$ Propose Block $\rightarrow$ "Wait for pre-commit from $+2/3$"

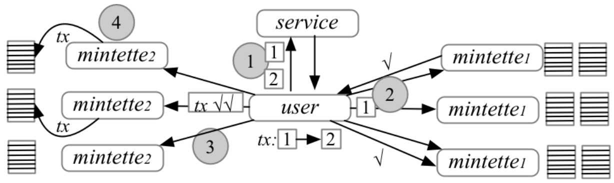


Fig. 44. The process of validating transactions; each mintette m_i is an owner of address i. In (1), a user learns the owners of each of the addresses in its transaction. In (2), the user collects approvals from a majority of the owners of the input addresses. In (3), the user sends boththe transaction and these approvals to the owners of transaction identifier. In (4), some subset of these mintettes add the transactions to their blocks [239].

$\rightarrow$ Commit. If some bad things/events happen (e.g., the timeout for a proposed block or invalid block), the validator submits a special vote called *Prevote nil*. If the validator does not receive enough pre-votes for the proposed block, it submits another special vote, called *Precommit nil*. If $+2/3$ pre-commits are not received within the pre-commit time period, the next round is initiated where a new proposer is selected, and the state machine repeats the above processes.

To ensure safety across rounds, the locking component of Tendermint should follow some rules on the blockchain height. One rule is that the validators must pre-vote the same block in the next round for the same blockchain height; and the other rule is that the unlocking operation is possible only when a newer block receives the proposed block in a later round for the same blockchain height. Under the assumption on the partial synchrony and locking rules, Tendermint can guarantee safety when less than one-third of validators exhibit Byzantine behaviors, and guarantee no fork occurs. Besides, Tendermint favors safety over availability (or liveness). For more details, interested readers can refer to the works [237] [238].

40) RSCoin - (blockchain): RSCoin is a cryptocurrency framework in which central banks maintain complete control over the monetary supply, proposed by Danezis and Meiklejohn in 2016 [239]. Although there exist centralized entities to control, it indeed relies on a distributed set of authorities, or *mintettes* (following the terminology of Laurie [240]), to reach a consensus and prevent double-spending in cryptocurrencies. The mintettes process the *lower-level blocks*, which form a potentially cross-referenced chain. The communication between committee members takes place indirectly through clients, and it also relies on the clients to ensure the completion of transactions. RSCoin follow the following processes. A client first gets signed “clearance" from the majority of the mintettes that manage the transaction inputs. Next, the client sends the transaction and the signed clearance to mintettes corresponding to transaction outputs. The mintettes check the validity of the transaction and verify signed evidence from input mintettes that the transaction is not double-spending any inputs. If all checks pass, the mintettes append the transaction to be included in the next block. The system operates in epochs: at the end of each epoch, mintettes send all cleared transactions to a central bank, which collates transactions into blocks that are then appended to the blockchain.

In more details, despite the centralization of monetary control, RSCoin framework still provides strong transparency, auditability, and scalability guarantees via the underlying verification and consensus processes. The key part to maintain the consistency is a distributed set of mintettes that are responsible for the maintenance of a transaction ledger. The mintettes collect transactions from users and collate them into blocks, much as is done with traditional cryptocurrencies. In traditional scenarios, the set of miners are neither known nor trusted mining ones. They have no choice but to broadcast a transaction to the entire network and rely on a proof-of-work to defend against Sybil attacks. While the mintettes in RSCoin are authorized by the central bank, and thus both known and trusted to some extend (providing accountability), thus, the underlying consensus can avoid a heavyweight consensus, and adopt a simplified version, e.g., Two-phase commit (2PC). Fig. 44 shows the processes on how a RSCoin's protocol validates transactions. By following the strict transaction verification processes and with the help of a central entity, RSCoin can prevent double-spending. To increase the performance (i.e., throughput), RSCoin adopts a sharding technology, randomly dividing transactions into different shards, so that each mintette is responsible for a subset of all transactions. This potentially increases the scalability. Interested readers can refer to Section V of [239].

However, client/user-driven atomic commit protocols are vulnerable to the DoS attack if the client stops participating and the inputs are locked forever. These systems assume that clients are incentivized to proceed to the unlock phase. Such incentives may exist in a cryptocurrency application where an unresponsive client will lose its own coins if the inputs are permanently locked, but do not hold for a general-purpose platform where inputs may have shared ownership. Besides, RSCoin relies on a two-phase commit protocol executed within each shard which, unfortunately, is not Byzantine fault-tolerant and can result in double-spending attacks by a colluding adversary.

41) ByzCoin - (blockchain): ByzCoin is a Byzantine consensus protocol that leverages scalable collective signing to achieve strong consistency, proposed by Kokoris-Kogias in 2016 [241]. ByzCoin provides strong consistency on Bitcoin while preserving the Bitcoin features on the open membership, scalability, and transaction rate. It employs proof-of-membership by dynamically forming hash power-proportionate consensus groups that represent recently successfully block miners, by a sliding window share over the proof-of-work. For consensus among the members, ByzCoin uses a PBFT with collective signing (CoSi [242]) to reduce both the costs of PBFT rounds and the costs for “light" clients to verify transaction commitment. CoSi is a collective signing protocol that efficiently aggregates hundreds or thousands of signatures. Even CoSi is not a consensus protocol, in ByzCoin, it makes PBFT's prepare and commits phases scalable, reducing the communication complexity from $O(n^2)$ to $O(n)$. ByzCoin replaces the direct MAC-authenticated communication among nodes with digital signatures, which allows for the indirect communication. This allows ByzCoin to scale with sparser tree-based communication patterns to optimize transaction commitment and verification under the normal operation while guaranteeing safety and liveness under Byzantine faults. Also, CoSi protocol can create an efficient compact multi-signature that can be verified with an aggregate public key as efficiently as an individual key [243].

In more details, Bitcoin's consensus algorithm provides only a probabilistic consistency guarantee, while practically, a strong consistency could offer more benefits. For example, all miners instantly agree on the validity of blocks, without wasting resources to resolve the potential forks; clients do not need to wait for an extended period to confirm that a transaction is committed; and it can provide the forward security (i.e., once a block has been appended to the blockchain, it is forever at the chain). ByzCoin uses the PBFT as its consensus protocol. To provide a strong consistency, ByzCoin needs to address several key challenges: open membership, scalability to hundreds of replicas, proof-of-work block conflicts, and transaction commitment rate. To address both the open membership and proof-of-

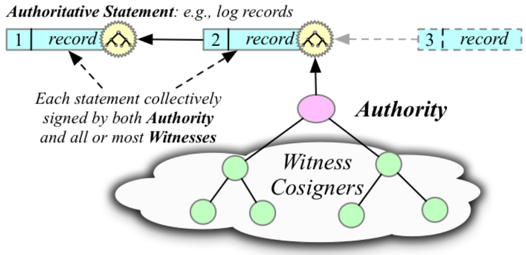


Fig. 45. CoSi protocol architecture [241].

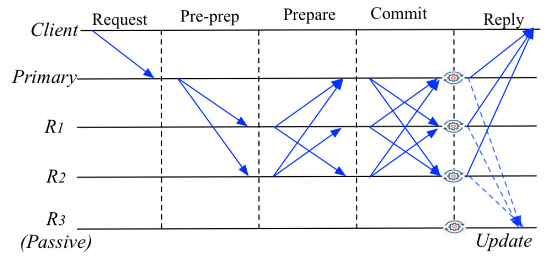


Fig. 46. Message patterns of consensus process in RePBFT [246].

work block conflicts, ByzCoin forms consensus groups dynamically from windows of recently mined blocks, giving recent miners shares or the voting power proportional to their recent commitment of hash power. To reduce transaction processing latency, ByzCoin adopts the idea from Bitcoin-NG [244] to decouple transaction verification from block mining. To achieve scalability, ByzCoin leverages CoSi to reduce the broadcast communication complexity of PBFT. Also, BzyCoin can achieve up to a near-optimal tolerance of f faulty group members among $3f + 2$ members in total. CoSi is one of the key contributions of ByzCoin.

CoSi is a protocol that performs an efficient signing by multiple entities. Fig. 45 shows a CoSi protocol architecture. CoSi utilizes a Schnorr signature [245] scheme to output a compact collective signature that is easily verifiable by clients. CoSi protocol in ByzCoin follows a tree-based communication structure to scale both computation and communication. For each message to be collectively signed, the leader then initiates a CoSi four-phase protocol round that requires two round-trips over the communication tree. This four-phase protocol consists of 1) *Announcement*, 2) *Commitment*, 3) *Challenge*, and 4) *Response*. For Announcement, the root of the communication tree initiates the protocol by multicasting the message to be signed. For Commitment, each node commits to a random secret and generates a public commit which it sends up the tree. Once receiving all commitments, non-leaf nodes aggregate the commit before sending it up the tree. For Challenge, the leader multicasts a collective challenge down the tree. For Response, the nodes compute their responses using their committed secret keys, which further aggregate the responses received from their children and send them further up the tree. In general, aggregation at every level allows nodes to check for the dishonest descendants, and the final collective signature can be verified by a Schnorr signature [243]. For more details on the optimization, interested readers can refer to the work [241].

42) ReBFT - (feature): Resource-efficient Byzantine Fault Tolerant (ReBFT) protocol is a scheme to minimizes the resource usage of a BFT system during a normal-case operation by keeping f replicas in a passive mode, proposed by Distler in 2016 [246]. In ReBFT, passive replicas neither participate in the agreement protocol nor execute client requests; instead, they are brought up to speed by the verified state updates provided by active replicas. However, when suspected or detected faults occur, passive replicas are activated in a consistent manner. In general, ReBFT reduces the resource footprint in the absence of faults, without losing its ability to ensure liveness in the presence of faults, which relies on two different modes of operation: normal case mode and fault handling mode. During a normal case operation mode, only a subset of the replicas are active, and the system can make progress as long as all replicas behave according to the specification. However, when faults are detected, it switches to a fault-handling mode by activating the remaining replicas, to tolerate the fault. Also, applying ReBFT does not require a BFT system to be completely redesigned from scratch.

In more details, ReBFT ensures that non-faulty replicas always maintain a consistent view on active and passive replicas. The active replicas keep passive replicas informed, by providing passive replicas with state updates. An update contains all state modifications that an active replica has performed as a result of the corresponding request. This means, although the f replicas are on standby, they must keep the same configuration as the $2f+1$ active replicas anytime. By doing so, the system can bring passive replicas up to speed, and when mode switching, the passive replicas can quickly be activated to switch a more resilient default BFT protocol to handle faults. More specifically, ReBFT uses $3f+1$ replicas for tolerating f faults. The authors show two use cases by adopting the ReBFT scheme to PBFT and MinBFT to achieve resource efficiency. Fig. 46 shows a message pattern of applying ReBFT to PBFT. One limitation is that, PBFT, MinBFT, and ReBFT do not support the faulty members rejoin the group, as when the number of faulty replicas is over f, the whole system stops functioning. Interested readers can refer to the work [246] for more details on both ReBFT-based PBFT and ReBFT-based MinBFT.

43) Solida - (blockchain): Solida is a decentralized blockchain protocol based on a reconfigurable Byzantine consensus augmented by the proof-of-work, proposed by Abraham et al. in 2017 [247]. It improves on Bitcoin in confirmation time, and provides the safety and liveness assuming the adversary controls less than one-third of the total mining power. By the improved confirmation time, it follows the concept of responsiveness in [248] on permissionless protocols. Specially, a protocol is responsive if it commits transactions (possibly in probabilistically) at a speed of the *actual* network delay, without being bottlenecked by hard-coded system parameters. Although Solida does no rely on the Nakamoto consensus for any part of the protocol, PoW still plays an important role to establish an imperfect Sybil-proof leader election oracle. However, the committee election and transaction processing are both done by a Byzantine consensus protocol in Solida. It uses a BFT protocol to resolve leader contention with certainty, rather than probabilistically in PoW of Nakamoto consensus. Besides the committee election and transaction processing, Solida provides a detailed principle to handle the reconfiguration process, since the protocol between two reconfiguration events is just a conventional Byzantine consensus. From a high-level perspective, Solida runs a Byzantine consensus protocol among a set of participants, called a committee, that dynamically change over time. At any time, only one committee member serves as the leader.

In more details, Solida considers a permissionless setting, in which participants use their public keys as pseudonyms and can join or leave the system at any time. The message transmission between honest participants should be within Δ time, and the ratio of Byzantine participants in the total of participants should be less than

$1/3$. Also, it assumes a delayed adaptive adversary, who takes time for the adversary to corrupt an honest participant. Besides, Solida has a rolling committee running a BFT to commit transactions and reconfiguration events into a ledger. For the reconfiguration process, Solida uses PoW to prevent Sybil attacks, as well as defense against selfish mining. Essentially, Solida has a big change compared with PBFT at an algorithmic level, from around-robin leader schedule to a potential interleaving between internal and external leaders. And Solida still has a total order among leaders to ensure its safety and liveness.

44) Guru - (middleware): Guru is a reputation model, which can be laid on top of various consensus protocols (e.g., PBFT or HoneyBadger BFT), proposed by Biryukov in 2017 [249]. In general, it ranks nodes based on the outcomes of the consensus rounds by a small committee, and adaptively selects the committee based on its current reputation. Also, it takes, as an input, the external reputation ranking. Guru can tolerate a threshold of malicious nodes, which can be up to slightly above $1/2$ since replication is a calculated criterion and it takes time and effort to earn reputation. Guru as a reputation module can be plugged into any round-based Byzantine agreement protocols. However, Guru does not care about if each round may reach a consensus or not, but the outcomes should be visible to all nodes. More specifically, Guru instructs the protocol to select the committee according to the reputation and maintains the reputation ranks, so that the nodes with a high reputation have a lower posterior probability of being malicious. And the prior probability of being malicious is given to the module, called external reputation. This external reputation typically is assigned once, e.g., nodes with trusted computing platforms may assign a higher external reputation than the one that does not have. The committee decision is made by a committee, which runs a round of a BFT protocol on the current set of transactions, and decides to either apply each of them or not. Typically, the committee is selected based on the current reputation that a node inherited or earned during the previous rounds. Also, assigning nodes to a committee requires some randomness, and the common randomness regularly comes from an external trusted source. The reputation module continuously observes whether the committee has reached a consensus to provide either rewards to honest nodes or penalties to malicious nodes. Besides, the authors also implemented a simulator for Guru from a high level.

45) RapidChain - (blockchain): RapidChain is a sharding-based public blockchain protocol that is resilient to Byzantine faults up to a $1/3$ fraction of its participants, proposed by Zamani et al. in 2018 [42]. It can achieve complete sharding of the communication, computation, and storage overhead of processing transactions without assuming any trusted setup. From a high-level perspective, RapidChain employs an optimal intra-committee consensus algorithm that can achieve high throughput via blockchain pipelining, a gossiping protocol for large blocks, and a provably secure reconfiguration mechanism to ensure robustness. RapidChain operates for public blockchain under Byzantine adversary. Traditional Byzantine consensus protocols can only work in a closed environment, where the set of participants is fixed and their identities are known to everyone via a trusted third party. If applied to an open setting, these Byzantine protocols may be easily compromised by Sybil attacks, e.g., the adversary can repeatedly rejoin malicious parties with fresh identities to gain significant influence on the protocol outcome. And most of these schemes assumed a static adversary who can select the set to corrupt only at the start of the protocol. In general, the PoW scheme allows the consensus protocol to impeded Sybil attacks by limiting the rate of malicious participants and provides a lottery mechanism to initiate the consensus process. It can help RapidChain to manage the membership of participants. At a high level, RapidChain partitions the set of nodes into multiple smaller groups of nodes, called committees, that operate in parallel on disjoint blocks of transactions and maintain disjoint ledgers. This partitioning process is also referred to as the *sharding*. By enabling parallelization of the consensus, sharding-based consensus can scale the throughput of the system proportional to the number of committees.

In more details, RapidChain assumes up to f replicas out of $3f+1$ replicas are controlled by a slowly-adaptive Byzantine adversary, the committee-election protocol samples a committee from the set of all nodes in a way that the fraction of corrupt nodes in the sampled committee is bounded by $1/2$ with a high probability. This is achieved by the Cuckoo rule [250] [251] for reconfiguration. And, there exists a *reference committee* responsible for driving periodic reconfiguration events between epochs. The intra-committee consensus requires that the members in a shard choose a local leader using the current epoch randomness, and then the leader sends the block to all members using a fast gossiping protocol that is based on an information dispersal algorithm (IDA) for large blocks. The members of a shard participate in a synchronous Byzantine consensus protocol [252], which allows RapidChain to obtain an intra-committee consensus with the optimal resiliency of $1/2$. This helps to achieve a total resiliency of $1/3$ with small committees. Another important aspect to guarantee RapidChain performing correctly is inter-committee communication.

For a inter-committee communication, the user in RapidChain does not attach any proof to a transaction. It lets the user communicate with any committee that routes the transaction to its output committee via the inter-committee routing protocol. RapaidChain considers a simple UTXO transaction $tx =< (I_1, I_2), O >$ that spends coins I_1, I_2 in shard S_1 and S_2, respectively, to create a new coin O belonging to shard S_3. The RapidChain engine executes tx by splitting it into three sub-transactions: $tx_a =< I_1, I'_1 >$, $tx_b =< I_2, I'_2 >$, and $tx_c =< (I'_1, I'_2), O >$, where I'_1 and I'_2 belong to S_3. tx_a and tx_b essentially transfer I_1 and I_2 to the output shard, which are spent by tx_c to create the final output O. All three sub-transactions are single-shard. In case of failures, when, for example, tx_b fails while tx_a succeeds, RapidChain sidesteps atomicity by informing the owner of I_1 to use I'_1 for future transactions, which has the same effect as rolling back the failed tx. The cross-shard transaction in RapidChain has largely relied on the inter-committee routing scheme which enables the users and committee leaders to quickly locate to which committees they should send their transaction. To achieve this, RapidChain builds a routing overlay network, at the committee level, which is based on a routing algorithm of Kademlia [253]. Specifically, each RapidChain committee maintains a routing table of $log(n)$ records which point to $log(n)$ different committees which are distance 2^i for $0 \leq i \leq logn - 1$ away.

For cross-shard (or inter-committee) transactions in RapidChain, one drawback is that, for each transaction, it creates three different transactions to exchange information among shards. This inherently increases the number of transactions to proceed, and the communication by sending the extra transactions back to its input committees also increases. It uses the committee's leader to produce these transactions without considering the status of a leader (e.g., malicious leader). Also, the input committees include the created new transactions into its leader. This behavior to some extent modifies the originality of transactions. Besides, the cross-shard transaction largely depends on

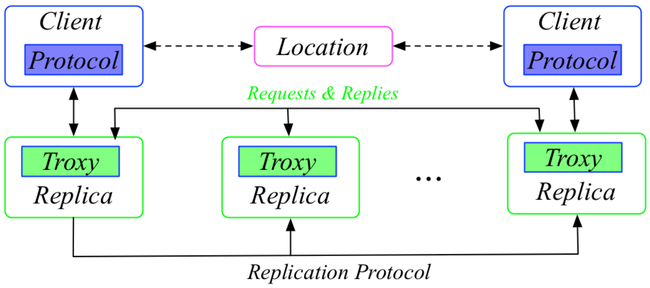


Fig. 47. Architecture of a Troxy-backed BFT system [254].

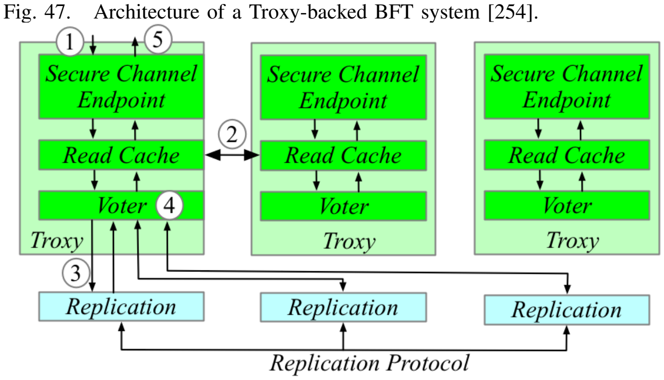


Fig. 48. Overview of Troxy components and their interactions [254].

the routing algorithm, which is a potential bottleneck.

46) Troxy - (middleware): Troxy is a trusted-proxy system that relocates the BFT-specific client-side functionality to the server side to make BFT *transparent* to legacy clients, proposed by Li et al. in 2018 [254]. To achieve this transparency, Troxy relies on a trusted subsystem built upon hardware protection enabled by Intel SGX. Also, Troxy reduces the replication cost of BFT for read-heavy workloads by offering an actively maintained cache that supports trustworthy read operations while preserving a consistency guarantees offered by the underlying BFT protocol. That means, with the help of trusted hardware, Troxy can guarantee the trustworthiness even the presence of Byzantine failures in surrounding replicas, and offer a trusted proxy to clients. Messages exchanged between Troxies and replicas are authenticated using common message certificates. Even with the immutability to arbitrary behaviors of replicas, the Troxy instance still has a chance to be crashed or is disconnected from its clients, which leads it unavailable. Typically, this scenario can be handled by a DNS round-robin or load-balancing appliances that enable a fail-over to another Troxy instance. The authors also proposed a solution that implements Troxy on top of Hybster, a hybrid BFT system that already features a trusted subsystem to reduce the number of replicas to $2f+1$. Nevertheless, Troxy can build an independent extension that can be applied to other hybrid systems featuring a trusted subsystem and traditional BFT agreement protocols.

In more details, Troxy acts as a representative of the client at the server-side and allows legacy client implementations to benefit from Byzantine fault-tolerate without requiring modifications. Due to the transparency to clients, Troxy does not incur additional network or processor usage at the client side. Fig. 47 shows an architecture of a Troxy-backed BFT system. A client is only required to establish a connection to a single Troxy instance, which then handles the communication with the replicas in the system for all of its clients. If a Troxy instance fails, the affected clients can re-establish their connections to the service as they would do in a traditional system, e.g., switching to different Troxies. Different from the replica components in the underlying replicated system, Troxies are trusted and assumed to only fail by crashing. To achieve this purpose, each Troxy runs inside a trusted subsystem, e.g., Intel SGX, which guarantees the integrity of the executed program codes. To protect the communication of a client with the server, a Troxy supports the establishment of secure channels using Transport Layer Security (TLS).

Fig. 48 shows an overview of the interactions among different Troxy components and with other system components outside the Troxy. When a client issues a request to the server via a secure channel, the Troxy first decrypts the message ((1)). It distinguishes two requests: read and write. For a read request, the Troxy executes the fast path ((2)), and in case of success, immediately returns the cached reply. For a write request or in case of a read-cache miss, the Troxy forwards the client request to its local replication logic to invoke a BFT agreement protocol ((3)). Once received the request, the BFT protocol distributes the request to other replicas in the system and ensures that all correct replicas execute all client requests in the same order. Once processing is finished in the BFT system, the replies from replicas will connect to the Troxy's voting component to determine the correct results by comparing the replies from different replicas ((4)). Once got an agreement, e.g., receiving $f+1$ matching replies from distinct replicas, the reply will be returned to the client ((5)) as this guarantees that at least one of the replies stems from a non-faulty replica and is therefore correct. Interested readers can refer to the work [254] for more details.

47)Thunderella - (theory): Thunderella is a paradigm for achieving state machine replication by combining a fast, asynchronous path with a (slow) synchronous "fall-back" path to provide an optimistic responsiveness, proposed by Pass and Shi in 2018 [255]. Thunderella is as robust as the best synchronous protocol, yet "optimistically" if a majority of its players are honest, the protocol "instantly" confirms transactions. This work is much theoretical, and the authors provide instantiations in both permissionless and permissioned settings. The key observation under Thunderella is to combine a centralized approach with a decentralized approach to achieve the scalability and speed of centralized approaches and the decentralized nature of the blockchain. In practice, Thunderella protocol can combine any "standard" blockchain, referred to as the slow chain, with the optimistic "fast-path". The fast path protocol is executed by a committee of stake-holders and coordinated by a central authority called the *Accelerator* (functioning as a "leader" in traditional leader-based BFT protocols), whose main job is to linearize transactions and data [256].

In more details, Thunderella follows the notion of *responsiveness* as proposed in prior work [257] to introduce the notion of optimistic responsiveness in synchronous settings. A consensus protocol is said to be a responsiveness protocol *iff* any transaction input to an honest node is confirmed in time that depends only on the *actual* network delay, but not on any a-priori known upper bound on the network delay. In Thunderella, the authors provide two kinds of delays: δ and Δ. The δ denotes the actual network delay and the Δ denotes an a-priori known upper bound of the network's delay where Δ is possible provided as input to the protocol. The Thunderella protocol allows synchronous protocols to commit responsively when some optimistic conditions are met. Thunderella is safe against up to one-half Byzantine faults. Moreover, if an accelerator and more than $3/4$ replicas are honest, and if they are on a "fast-path", then replicas can commit responsively in $O(\delta)$ time, otherwise, the protocol falls back to a "slow-path", which has a commit latency that depends on the delay Δ.

Practically, the optimistic responsiveness of Thunderella requires replicas to know which of the two paths they are in, and explicitly switch between them [258]. If, at some points, the optimistic conditions cease to be met, the replicas switch to a slow-path. However, when the optimistic conditions start to hold again, the players switch back to the fast-path. Besides, Thunderella uses Nakamoto's protocol or the Dolev-Strong protocol [13] as its slow-path. This result that the slow-path, as well as the switch between two paths, is extremely, requiring $O(k\Delta)$ and $O(n\Delta)$ latency respectively (where k is a security parameter).

A follow-up on the optimistic responsiveness is presented in Hybrid-BFT by Momose et al. in 2020 [259]. Under the work of Thunderella, the authors state that a protocol makes a decision with latency on the order of an actual network delay δ if the number of actual faults is significantly smaller than f, which is the worst-case allowed. However, in the presence of f faults, the protocol incurs $O(k\Delta)$ (k is a security parameter) or $O(f\Delta)$ latency, which is far from the optimal case. To provide a strong availability, the latency should also be as short as possible in the presence of f faults. Without optimistic responsiveness, some protocols incur latency that is close to optimal $\Delta+O(\delta)$ [260] or nearly optimal $2\Delta+O(\delta)$ [261]. One question raised in Hybrid-BFT is: " Can a Byzantine consensus protocol simultaneously achieve optimistic responsiveness as well as optimal $\Delta + O(\delta)$ latency in the presence of f faults?" The Hybrid-BFT can achieve both properties simultaneously. To show its findings on optimistic responsiveness and optimal latency, Hybrid-BFT presents a simple leader-based BFT protocol as a practical application. Even while being able to rotate leaders after every decision, the proposed protocol can simultaneously achieve an average latency of 1) 3δ under the optimistic condition and 2) $1.5\Delta + O(\delta)$ (or $3\Delta + O(\delta)$) in the presence of faults, which is more than a factor of two better than existing rotating-leader BFT protocols. Interested readers can refer to the work [259] for detailed proofs.

48)Chainspace - (blockchain): Chainspace is a recently proposed, sharded smart contract platform with privacy built-in by design, proposed by Al-Bassam in 2018 [262]. To enable scalability on Chainspace, nodes are organized into shards that manage the state of objects, keep track of their validity, and record transactions committed or aborted. These nodes ensure that only valid transactions, consisting of encrypted or committed data, along with the zero-knowledge proofs that assert their correctness, end up on their shards of the blockchain. The nodes are typically required to communicate with the other shards to decide whether to accept or reject a transaction via an inter-shard consensus. Instead of a client-driven approach, Chainspace runs an atomic commit protocol collaboratively between all the concerned committees. This is achieved by making all committees act as resource managers for the transactions they manage. To do so, Chainspace proposes a protocol called *Sharded Byzantine Atomic Commit* or *S-BAC*, which basically combines the existing Byzantine agreement and atomic commit protocols in a novel way. Byzantine agreement securely keeps a consensus on a shard of $3f + 1$ nodes in total, containing up to f malicious nodes. Atomic commit runs across all shards that contain objects on which the transaction relies. The transaction is rejected unless all of the shards accept to commit the processed transaction.

In more details, the intra-shard consensus algorithm assumes an

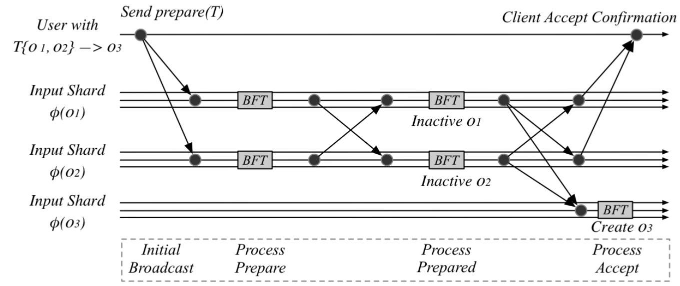


Fig. 49. *S-BAC* for a transaction T with two inputs (o_1, o_2) and one output object (o_3). The user sends the transaction to all nodes in shards managing o_1 and o_2. The BFT-Initiator takes as a leader in sequencing T, and emits 'prepared(accept, T)' or 'prepared(abort, T)' to all nodes within the shard. Next, the BFT-Initiator of each shard assesses whether overall 'All proposed(accept, T)' or 'Some proposed(abort, T)' holds across shards, sequences the accept (T, *), and sends the decision to the user. All cross-shard arrows represent a multicast of all nodes in one shard to all nodes in another [262].

asynchronous setting and relies on the S-BAC protocol to handle transactions. Essentially, S-BAC protocol consists of two primitive protocols: Byzantine agreement and atomic commit. The Byzantine agreement ensures that all honest members of a shard of size $3f + 1$ agree on a special common sequence of actions. Chainspace utilizes MOD-SMART [23] implementation of PBFT to provide an optimal number of communication steps. Specially, this is achieved by replacing broadcast with a special leader-driven *Validated and Provable Consensus (VP-Consensus)*. Atomic commit runs across all shards managing objects relied upon by a transaction to ensure consistency among shards. For example, if a single shard rejects a transaction, then all agree that the transaction is rejected. Chainspace utilizes a two-phase commit protocol [263], composed with an agreement protocol to achieve consistency. S-BAC composes the above primitives to ensure that shards process safely and consistently all transactions. Fig. 49 shows an example of the S-BAC protocol to commit a single transaction with two inputs and one output. It consists of four phases: initial broadcast (prepare), process prepare, process prepared (accept or abort), and process accept. In each shard, there exists a designated node, called BFT-Initiator, to assist the whole process. Specially, the BFT-Initiator derives the composed S-BAC protocol by sending "prepare" and then "accept" messages to reach a BFT consensus within and across shards. Also, it is responsible for broadcasting consensus decisions to relevant parties. Besides, Chainspace supports a two-phase process to recover from a malicious BFT-Initiator that suppresses transactions. Interested readers can refer to Chainspace [262] for more details.

49) OmniLedger - (blockchain): OmniLedger is a scale-out distributed ledger that preserves a long-term security under permissionless operations, proposed by Kokoris-Kogias et al. in 2018 [264]. It ensures security and correctness by using a bias-resistant public-randomness protocol for choosing large, statistically representative shards to process transactions. Essentially, OmniLedger uses a Byzantine shard atomic commit (Atomix) protocol to atomically process transactions across multiple committees, such that each transaction is either committed or aborted. Since both deploying an atomic commit protocol and running a BFT consensus are unnecessarily complex, Atomix uses a lock-then-unlock process to achieve consistency. OmniLedger intentionally keeps the shards' logic simple and makes any direct shard-to-shard communication unnecessary by tasking the client with the responsibility of driving the unlock process while permitting any other party (e.g., validators or even other clients) to fill in for the client if a specific transaction stalls after being submitted for processing. Atomix takes a three-step (*initialize/lock/unlock*) protocol to deal with cross-shard UTXO transactions. More specifically, the client first gossips the cross-shard transaction to all their input shards. Then, OmniLedger takes a two-phase approach to achieve the atomic commit, in which each input shard first locks the corresponding input UTXO(s) and issues a proof-of-acceptance, if the UTXO is valid. The client collects responses from all input committees and issues an “unlock to commit" to the output shard.

In more details, OmniLedger assumes a slow-adaptive adversary that can corrupt up to a $1/4$ fraction of the nodes at the beginning of each epoch. To allow new participants to join in the protocol, OmniLedger runs a global reconfiguration protocol at every epoch. Under the assumption of synchronous channels, the protocol generates identities and assigns participants to committees using a slow identity blockchain protocol. In each epoch, OmniLedger requires fresh randomness, by using a bias-resistant random generation protocol, for an unpredictable leader election process. The random generation protocol used in OmniLedger is based on a verifiable random function (VRF) which is similar to the lottery algorithm in Algorand. The intra-committee consensus protocol assumes the existence of partially synchronous channels to achieve a ByzCoin-like fast consensus, where the committee is further divided into smaller groups using epoch randomness. However, the design of ByzCoin has several security/performance issues, e.g., falling back to all-to-all communication in the Byzantine setting. The detailed processes and discussions can refer to OmniLedger [264].

As RapidChain [42] pointed out, there are several challenges that OmniLedger leaves unsolved. OmniLedger can only tolerate $1/4$ of total corruptions; and from the simulation results, the protocol can only achieve low latency when $1/8$ of total replicas corrupted. OmniLedger requires $O(n)$ pre-node communication as each committee has to gossip multiple messages to all n nodes for each block of the transaction, also it requires a trusted setup to generate an initial unpredictable configuration to seed the VRF in the first epoch. Besides, OmniLedger requires the client to participate actively in cross-shard transactions, and this may be a strong assumption for lightweight clients.

50) Dynamic Byzantine Protocol Adaption - (feature): Dynamic adaption for Byzantine consensus protocols explores the mechanism that allows the algorithms to be adapted or changed at runtime to optimize the system to the current conditions, proposed by Carvalho et al. in 2018 [265]. The authors conducted studies on how different dynamic adaption techniques proposed for crash failure models can be applied to in the presence of Byzantine faults. Also, the authors presented a comparative study of the performance of these switching algorithms in practice. Most existing fault-tolerant solutions are customized, and no one-size-fits-all solution for a range of extended operating conditions. It is critical to develop some mechanisms to envelop the underlying consensus processes and provide a uniform interface to end users. Dynamic adoption of consensus algorithms would help to resolve this kind of situation. Typically, adaptive protocols can be classified into two categories: those using a *single reconfiguration consensus protocol* and those using *multiple consensus protocols*. Using a reconfigurable consensus algorithm is to develop an adaptation-aware monolithic protocol, which incorporates all behaviors that can be activated at runtime. For example, the work [227] presented a reconfigurable variant of PBFT that allows for online changes in the batching size and timeouts to adapt to changes

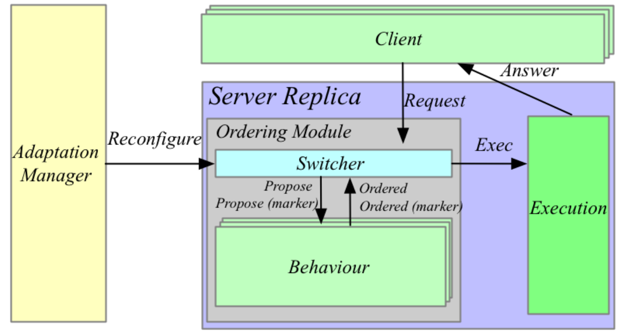


Fig. 50. System architecture of dynamic BFT adaption [265].

in the workload characterization. This approach typically requires coordinating the reconfiguration of different replicas to ensure they always operate in compatible environments. While using multiple consensus algorithms as black-boxes is a more modular solution, consisting of implementing different behaviors using independent protocols. Typically, this approach requires some mechanisms to switch between independent protocols, e.g., a *switcher* enabling switching processes among two or more protocols. One of the advantages of this approach is that it allows any protocol to be integrated without involving major efforts, as it requires no adaptive features from the consensus protocols. And the switching process can be facilitated if the protocols export an interface that allows a switcher to deactivate the protocol placing it in a quiescent state [266].

Based on the above state-of-the-art literature, the authors proposed a BFT-SMART based instantiation for dynamic adaptation. It assumes a partially synchronous network, and there exists a component, called the *switcher*, that is responsible for forwarding the requests to one or more of these protocols in multiple consensus scenarios. There is an external component, called an *adaptation manager*, that decides which protocol or configuration is going to be used at any moment. The adaptation manager can send a command to all switchers to start the adaptation, and these adaptation commands are ordered and identified with a monotonically increasing *id*. Typically, the implementation of the adaptation manager is orthogonal to the deployed consensus protocols, and the manager itself is also replicated and each switcher only initiates switching when it receives an identical command from a quorum of adaptation manager replicas. Fig. 50 shows an abstract of the system architecture for BFT-SMART instantiation. It makes a set of extensions to BFT-SMART, e.g., integrated a switcher to mediate between clients and the supported consensus algorithms, integrated a Fast-SMART [134], and developed a stoppable version of MoD-SMART [23] and Fast-SMART. Interested readers can refer to Section 4 in [265].

51) Policy-based BFT Adaptation - (feature): Policy-based adaptation for BFT protocols explores a different alternative to support the system changes by using a policy language, proposed by Bravo et al. in 2018 [267]. This alternative consists of specifying not only the system configuration, but also the fault-handling behavior, and how the system adopts to these changes in a workload, in a policy language, that is processed externally to the managed systems. Typically, different configurations have different performances in face of variable workloads. For example, even the basic functionality of the system may vary depending on the type and scope of faults that one is considering. A policy-based adaptation offers a strong separation of these concerns between the basic mechanisms offered by a system and the policies defined by the system administrator to

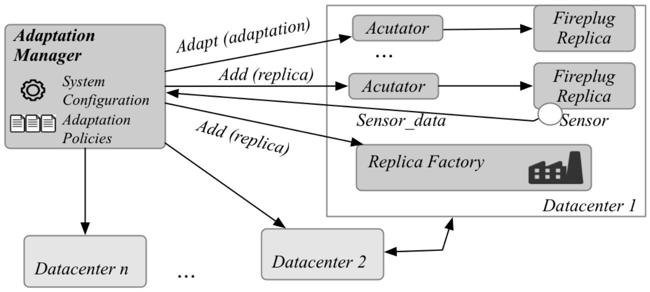


Fig. 51. Overview architecture of Policy-based adaption [267].

meet target performance and dependability requirements. The authors presented an approach by applying it to configure and dynamically adopt Fireplug [268], a flexible architecture to build robust geo-replicated graph databases. From a high-level perspective, Fireplug is a Byzantine fault-tolerant distributed transactional graph database management system. It is designed to run in one or more datacenters, in which each datacenter runs multiple data nodes or servers with each node running an instance of the graph database. Typically, Fireplug can support two types of transactions: read-only transactions and update transactions.

More specifically, the original Fireplug prototype did not support any policy-based configuration or adaptation, with all configurations performed at the building time using compilation options. To support policy-based schemes, Fireplug is required to implement some new components, e.g., adaptation manager, actuators, and replica factory. Fig. 51 shows an overall architecture of the policy-based Fireplug. The adaptation manager is in charge of executing a configuration and an adaptation policy that is provided by the operator in a high-level form. Multiple policies can be installed in the adaptation manager to handle different aspects of the system operations. It requires that all policies should be consistent, e.g., no two installed policies conflicting, and requires the extensibility to allow the manager to select among alternative policies. Typically, the adaptation manager is fed by a number of sensors that provide information about the current state of the system. Each data node and each client proxy are augmented with an *actuator*, a component that receives commands from the adaptation manager and mutates the behavior of the Fireplug components. Also, it requires a special *replica factory* deployed in each datacenter. The factory is in charge of launching a new replica in its local datacenter, upon the requests from the adaptation manager. The sensors provide inputs to the adaptation manager, e.g., information of operational conditions of Fireplug. While, actuators provide the adaptation manager module with multiple actions that can be called. These actions include the operations to activate/deactivate a replica, switch the read-only/update transactional protocol, and join/leave a local or global group. Interested readers can refer to Sections IV and V of [267] on detailed adaptation and policies.

52) Subquadratic Byzantine Agreement - (theory): Subquadratic Byzantine agreement (BA) protocols are a group of consensus protocols which can achieve agreements in less quadratic communication complexity (e.g., whose complexity growth mush slower than $O(n^2)$), proposed by Abraham et al. in 2019 [269]. The presented work focuses on the theoretical results, specifically, two impossibilities. Although many Byzantine agreement protocols have improved communication complexity, most of them are based on some strong assumptions, e.g., the assumptions on a static adversary. And a few existing works have shown how to achieve subquadratic BA under an *adaptive* adversary. Even under the assumption of an adaptive adversary, they commonly limit the ability of adaptivity. For example, if an honest node sends a message and then gets corrupted in some rounds, the adversary cannot erase the message that was already sent, and this assumption commonly refers to that an adversary cannot perform “after-the-fact-removal". This means the adversary is unable to erase the honest messages sent even the honest nodes (who sent this message) compromised. The proposed work first proves that disallowing after-the-fact removal is necessary for achieving subquadratic-communication BA, and then presents a subquadratic binary BA construction, under the after-the-fact removal assumption, that achieves near-optimal resilience and expected constant rounds under standard cryptographic assumptions and PKI. Many natural $\Omega(n^2)$-communication BA protocols [270] [13] [271] can be proven secure even if the adversary is capable of after-the-fact removal. And this kind of adversary are commonly referred as a *strongly adaptive adversary*. However, they did not show if it can achieve this with a subquadratic communication.

In more details, the authors show the main theorems on three aspects: the necessity of disallowing “after-the-fact" removal, the near-optimal subquadratic BA minimal assumption, and the necessity of setup assumption. The proofs of these theorems are a little bit lengthiness. We excerpt these results only. Theorem 1 is about the impossibility of BA with subquadratic communication, w.r.t. a strong adaptive adversary. It states: any (possibly randomized) BA protocol must in expectation incur at least $O(f^2)$ communication in the presence of a strongly adaptive adversary capable of performing after-the-fact removal, where f denotes the number of corrupted nodes. Theorem 2 is about near-optimal subquadratic BA. It states: assuming standard cryptographic assumptions and a public-key infrastructure (PKI), for any constant $0 < \epsilon < 1/2$, there exists a synchronous BA protocol with expected $O(k)$ multicast complexity, expected $O(1)$ round complexity, and $exp(-\omega(k))$ error probability that tolerates $f < (1-\epsilon)n/2$ adaptively corrupted players out of n players in total. Theorem 3 is about the impossibility of sublinear multicast BA without setup assumptions. It states: in a plain authenticated channel model without setup assumptions, no protocol can solve BA using C multicast complexity with probability $p > 5/6$ under C adaptive corruptions. Interested readers can refer to the work [269] for more details.

53) Probabilistic Byzantine Reliable Broadcast - (theory): Probabilistic Byzantine broadcast is a kind of analysis in Byzantine setting, and a scalable probabilistic Byzantine reliable broadcast protocol was proposed by Guerraoui et al. in 2019 [272]. Existing broadcast protocols for Byzantine agreement protocols scale poorly, as they typically build on some *quorum systems* with strong intersection guarantees, which results in linear per-process communication and computation complexity. The authors in [272] generalized the Byzantine reliable broadcast abstraction to a probabilistic setting, allowing each of its properties to be violated with a fixed arbitrarily small probability. Then, it leverages these relaxed guarantees in a protocol to replace the quorums with stochastic samples. In general, samples can be significantly smaller in size, which leads to a more scalable design. The authors designed a Byzantine reliable broadcast protocol with logarithmic per-process communication and computation complexity. Also, the authors introduced a general technique called *adversary decorators*, which allows them to make claims about the optimal strategy of the Byzantine adversary without imposing any additional assumptions. Besides, the authors introduced Threshold Contagion, a model of message propagation through a system with Byzantine processes.

In more details, the authors presented a probabilistic gossip-based Byzantine reliable broadcast algorithm having $O(logN)$ per-process communication and computation complexity at the expense of $O(logN/loglogN)$ latency. The probabilistic algorithm, Contagion, allows each property of Byzantine reliable broadcast to be violated with an arbitrarily small probability ϵ. And ϵ scales sub-quadratically with N, and decays exponentially in the size of the samples. The Contagion algorithm incrementally relies on two sub-protocols: Murmur and Sieve. Essentially, Murmur is a probabilistic broadcast algorithm that uses simple message dissemination to establish validity and totality, and each correct process relays the sender's message to a randomly picked gossip sample of other processes. Sieve is a probabilistic consistent broadcast algorithm (built upon Murmur) that guarantees consistency, e.g., no two correct processes deliver different messages, and each correct process uses a randomly selected echo sample. While Contagion is a probabilistic Byzantine reliable broadcast algorithm that guarantees validity, consistency, and totality, whose sender uses Sieve to disseminate a consistent message to a subset of the correct processes. Interested readers can refer to the details on [272].

A similar analysis work on the round complexity of randomized Byzantine agreement (BA) was proposed by Cohen et al. in 2019 [273]. In the work, the authors proved lower bounds on the round complexity of randomized Byzantine agreement protocols, bounding the halting probability of such protocols after one and two rounds. The authors proved three important bounds, as excerpted from the work [273]. 1) Randomized BA protocols resilient against $n/3$ [resp., $n/4$] corruptions terminate (under attack) at the end of the first round with probability at most $O(1)$ [resp., $1/2 + o(1)$]. 2) Randomized BA protocols resilient against $n/4$ corruptions terminate at the end of the second round with probability at most $1 - \Theta(1)$. And, 3) For a large class of protocols, including all BA protocols used in practice) and under a plausible combinatorial conjecture, BA protocols resilient against $n/3$ [resp., $n/4$] corruptions terminate at the end of the second round with probability at most $o(1)$ [resp., $1/2+O(1)$]. The above three bounds hold even when the parties use a trusted setup phase, e.g., a public-key infrastructure (PKI).

54)SBFT - (feature): SBFT is a Byzantine replicated system that addresses the challenges of scalability, decentralization, and global geo-replication, proposed by Gueta et al. in 2019 [65]. Essentially, SBFT uses a combination of four components: using collectors and threshold signatures to reduce communication to linear, using an optimistic fast path, reducing client communication, and utilizing redundant servers for the fast path. Also, SBFT implements a correct dual-mode view change protocol that allows to efficiently run either an optimistic fast-path or a fallback slow path without incurring a view change to switch between nodes. In general, scaling BFT replication to tolerate tens of malicious nodes requires to re-think BFT algorithms and re-engineer them for a high scale. SBFT is a BFT system optimized to work over a group of hundreds of replicas in a world-scale deployment. The authors of SBFT provide detailed evaluations in a world-scale geo-replicated deployment with 209 replicas withstanding $f = 64$ Byzantine failures. It shows how different algorithmic components that SBFT used increase its performance and scalability. Also, the authors show that SBFT simultaneously provides almost $2X$ better throughput and about $1.5X$ better latency relative to a highly optimized system that implements the PBFT protocol.

In more details, SBFT assumes a standard asynchronous BFT system model, and it distinguishes two modes: a synchronous mode when the adversary can control up to f Byzantine nodes but with a known bounded delay between non-faulty nodes, and a common mode when the adversary only controls $c \leq f$ nodes that can only crash or act slow and a known bounded delay between non-faulty nodes. Under $n = 3f + 2c + 1$ replicas and the common mode, SBFT obtains safety, liveness, and linearity. Fig. 52 shows a schematic message patterns for $n = 4, f = 1, c = 0$. To achieve this, SBFT adds some components/optimization. Optimization 1): from PBFT to linear PBFT. SBFT reduces all-to-all communication patterns to a linear communication pattern by using a *collector*, whose function is to collect requests from each replica and broadcast the collected requests to everyone. Also, by using threshold signatures, one can reduce the outgoing collector message size from linear to constant. Also, SBFT pushes the collector duty to replicas in a round-robin manner. Optimization 2): adding a fast path. SBFT, as in Zyzzyva, allows for a faster agreement path in *optimistic executions*, where all replicas are non-faulty and the network is synchronous. SBFT implements a practical dual-mode view change protocol, which can seamlessly switch between paths and adjust to network/adversary conditions (without an actual view change). Optimization 3): reducing client communication from $f + 1$ to 1. Threshold signature schemes can be used to reduce the number of messages a client need to receive and verify. In SBFT, each client needs only one message, containing a single public-key signature for request acknowledgments. SBFT reduces the per-client linear cost to just one message by adding a phase that uses an *execution collector* to aggregate the execution threshold signatures and sends each client a single message carrying one signature. Optimization 4): adding redundant servers to improve resilience and performance. SBFT allows a fast path to tolerate up to a small number c of crashed or straggler nodes out of $n = 3f+2c+1$ replicas, to make the fast path more prevalent. SBFT only falls back to the slower path if there are more than c faulty replicas.

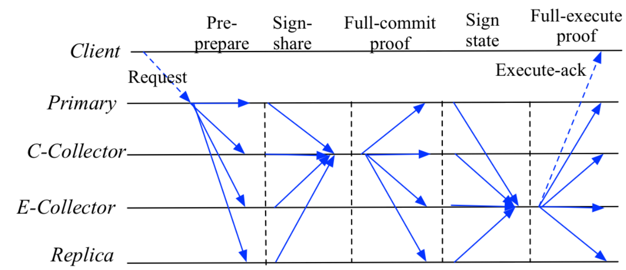


Fig. 52. Message patterns of SBFT [65].

Besides, SBFT uses a short Boneh-Lynn-Shacham (BLS) signatures [274] scheme which has security that is comparable to 2048-bit RSA signatures but are just 33 bytes long. It can improve communication and computation efficiency.

55) Flexible BFT - (feature): Flexible BFT is an approach for BFT consensus protocols solution to provide stronger resilience and diversity, proposed by Malkhi et al. in 2019 [275]. The stronger resilience involves a new fault model, *alive-but-corrupt* (aka. *a-b-c*) faults, in which replicas may arbitrarily deviate from the protocol in an attempt to break the safety of the protocol. However, if they cannot break the safety, they will not try to prevent the liveness of the protocol. By considering alive-but-corrupt faults into a Byzantine setting, Flexible BFT is resilient to higher corruption levels than that in a pure Byzantine fault model. The diversity of Flexible BFT provides a solution whose protocol transcript is used to draw different commit decisions under diverse beliefs. With that separation, the same Flexible BFT solution supports synchronous and asynchronous beliefs, and varying resilience threshold combinations of Byzantine and alive-but-corrupt faults. To achieve these goals, from a technical level, Flexible BFT achieves the above results using two mechanisms: a synchronous BFT protocol and Flexible Byzantine Quorums. For the synchronous BFT protocol, only the commit step requires knowing the network delay bound and thus replicas execute the protocol without any synchrony assumption; for the Flexible Byzantine Quorums, it dissects the roles of different quorums in existing consensus protocols.

In more details, most existing protocols are challenged if these underlying settings differ from the ones they are designed for. For example, optimal-resilience partially synchronous solutions break (i.e., lose safety and liveness) if the fraction of Byzantine faults exceeds $1/3$. And many robust protocols are based on crashed faults. The Flexible BFT makes a more rational fault model, called alive-but-corrupt faults. The rationale is based on the observation that violating safety may provide some attacker gains (e.g., a double-spending attack), but preventing liveness usually does not. Also, the Flexible BFT provides a certain separation between the fault model and the protocol. Its design approach builds a protocol whose transcript can be interpreted by learners with diverse beliefs, who draw different consensus commit decisions based on their beliefs. The Flexible BFT guarantees safety and liveness for all learners that have correct beliefs. Each learner can specify (i) the fault threshold it needs to tolerate, and (ii) the message delay bound, if any, it believes in. This separation design philosophy provides several advantages. One thing is that different learners may naturally hold different assumptions about the system. For example, some learners may be more cautious and require a higher resilience than others, some learners may believe in synchrony while others do not. Also, a leaner may update its assumptions based on certain events it observes. For example, if a learner receives votes for conflicting values, it may indicate an attempt at attacking safety, and the learner can start requiring more votes than usual. The learners typically have different assumptions and hence different commit rules. And the Flexible BFT provides some mechanisms to handle these conflicts on different learners, e.g., via a “recovery" mechanism.

Besides, the Flexible BFT utilizes a synchronous BFT protocol and quorum-based mechanism. The adopted synchronous BFT protocol on replicas execution is at network speed (e.g., even in an asynchrony setting). This allows learners in the same protocol to assume different message delay bounds and commit at their own pace, which separates the timing assumption of replicas from timing assumptions of learners. This separation process is only available under some conditions: the action of committing is only carried out by learners, not by replicas. The other technique involves a breakdown of the different roles that quorums play in different steps of partially synchronous BFT protocols. Overall, by separation process, the Flexible BFT tolerates a fraction of combined (Byzantine and a-b-c) faults beyond existing resilience bounds. Interested readers can refer to the work [275] for more details.

56) VSSR BFT - (feature): VSSR is an acronym of Verifiable Secret Sharing with Share Recovery, which can provide efficient services for BFT protocols, proposed by Basu et al. in 2019 [276]. Typically, BFT SMR protocols fail to provide the privacy guarantee, and a natural way to add privacy guarantees is via the secret-share state instead of fully replicating it. However, incorporating a secret shared state into traditional BFT SMR protocols presents unique challenges. For example, the network model of asynchrony in BFT protocols makes verifiable secret sharing (VSS) unsuitable. Meanwhile, fully asynchronous VSS (AVSS) is unnecessary as well since the BFT algorithm provides a broadcast channel. The authors present to use VSS with share recovery scheme to incorporate secret shared state into a BFT engine. The authors also present a VSS with a share recovery solution, KZG-VSSR, in which a failure-free sharing incurs only a constant number of cryptographic operations per replica. Also, the authors present a way to efficiently integrate any instantiation of VSSR into a BFT replication protocol while incurring only constant overhead.

In more details, VSRR is a framework that, given a VSS scheme with certain properties, adds share recovery with only a constant factor overhead from the original VSS scheme. The authors present two ways to instantiate VSSR: 1) KZG-VSSR, which uses Kate et al.'s secret sharing scheme [277] to instantiate a VSS that has constant time share verification, and 2) Ped-VSSR, which uses Pedersen's secret sharing scheme [278]. The secret sharing scheme in Pedersen only provides linear time share verification and recovery, uses some cheaper cryptographic operations, and is faster for smaller clusters. The proposed framework is based on the recovery polynomial, which is a single polynomial that encodes recovery information for f shares. Thus, by only sharing a small, constant number of additional polynomials, the client can enable all $3f + 1$ shares to be recovered. A generic VSSR BFT scheme works in constant per sharing in a failure-free case. However, in the worst case, e.g., when there are f participant failures or the leader is Byzantine, the overhead is still quadratic, which incurs a linear overhead when incorporated into BFT replicas. While, the optimized VSSR schemes, e.g., KZG-VSSR and Ped-VSSR, incur only constant overhead. Interested readers can refer to the work [276] for more details.

57) Stellar - (blockchain): Stellar is a global payment network that can directly transfer digital money in the world in seconds by a quorum-based Byzantine agreement protocol, proposed by Lokhava et al. in 2019 [279]. To achieve this, Stellar uses a secure transaction mechanism across untrusted intermediaries, using a Stellar Byzantine agreement protocol (SCP). With SCP, each institution specifies other institutions with which to remain in the agreement, through a global interconnectedness of the financial system, the whole network then agrees on atomic transaction spanning arbitrary institutions, and this is with no solvency or exchange-rate risk from intermediary asset issuers or market makers. Essentially, Stellar is a blockchain-based payment network specifically designed to facilitate innovation and competition in international payments. It want to achieve three goals: 1) *open membership*, 2) *issuer-enforced finality*, and 3) *cross-issuer atomicity*. For the open membership, anyone can issue currency-backed digital tokens that can be exchanged among users; for the issuer-enforced finality, a token's issuer can prevent transactions in the token from being reversed or undone; and for the cross-issuer atomicity, users can atomically exchange and trade tokens from multiple issuers. There are different ways to achieve a separated or two combined goals, but not all. For example, financial companies such as Paypal and Venmo can easily achieve the first two goals; using a closed system can achieve the goals 2 and 3; while a mined blockchain can achieve the goals 1 and 3. Stellar network can achieve these three goals at once.

In more details, to achieve these goals, Stellar relies on two aspects: supporting efficient markets between tokens from different issuers and a Byzantine agreement protocol SCP. Anyone can issue a token, the blockchain provides a built-in orderbook for trade between

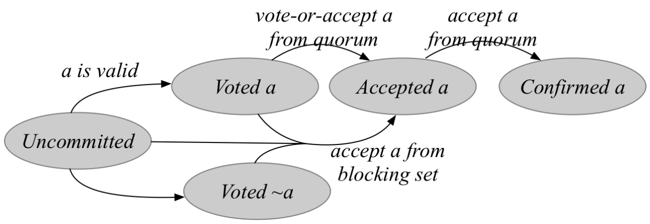


Fig. 53. Storage of federated voting [279].

any pair of tokens. And, users can issue *path payments* that atomically trade across several currency pairs while guaranteeing an end-to-end limit price. Via SCP, the token issuers designate some specific *validator* servers to enforce transaction finality. The issuer knows exactly which transactions have occurred and avoids the risk of losses from blockchain history reorganization. The key aspect under SCP is that most asset issuers benefit from liquid markets and want to facilitate atomic transactions with other assets. In general, the validator administrators configure their servers to agree with other validators on the exact history of all transactions on all assets. By using a trusted chain (e.g., one validator trusts another validator), agreement relationships can be transitively connected to the whole network, and SCP guarantees global agreement, making it a global Byzantine agreement protocol with open membership. We can consider the construction of a trusted relationship as a federated voting process. Fig. 53 shows some key stages of a federated voting. By voting on transactions, it can establish an agreement among untrusted validators. Thus, essentially, SCP is a quorum-based Byzantine agreement protocol, and quorums emerge from the combined local configuration decisions of individual nodes. The nodes only recognize quorums to which they belong themselves, and only after learning the local configurations of all other quorum members. By using this way, SCP can inherently tolerate heterogeneous views of what nodes exist, and nodes can join and leave unilaterally without the need for a view change protocol to coordinate the memberships. Interested readers can refer to the work [279] for more details.

58) DISPEL - (feature): DISPEL is an acronym of Distributed Pipeline Byzantine SMR that allows any node to distributedly start new consensus instances whenever they detect sufficient resources locally, proposed by Voron and Gramoli in 2019 [280]. This work is mostly based on the observation of important causes of performance limitation, like *head-of-line (HOL)* blocking [281], that delays the subsequent packet delivery due to a TCP packet loss. And DISPEL tries to design a distributed technology to mitigate this situation. Essentially, DISPEL balances its quadratic member of messages (e.g., in PBFT) onto as many routers between distributed replicas to offer high throughput with reasonably low latency. One key innovation of DISPEL is its distributed pipeline, a technique adapted from the centralized pipelining of micro-architectures to the context of SMR to leverage unused resources. Different from the centralized pipelining, the distributed pipelining maximizes the resource usage of distributed replicas by allowing them to decide locally when to spawn new consensus epochs. Each replica in DISPEL detects idle network resources at its network interface (NIC) and sufficient available memory spawns a new consensus instance in a dedicated pipeline epoch. Also, the distributed detection can leverage links of heterogeneous capacity like inter-datacenters vs. intra-datacenters communication links. Two main contributions are: 1) by removing the limitation of HOL blocking, 2) by enabling distributed pipelining. For the HOL blocking, DISPEL identified the HOL blockchain as the

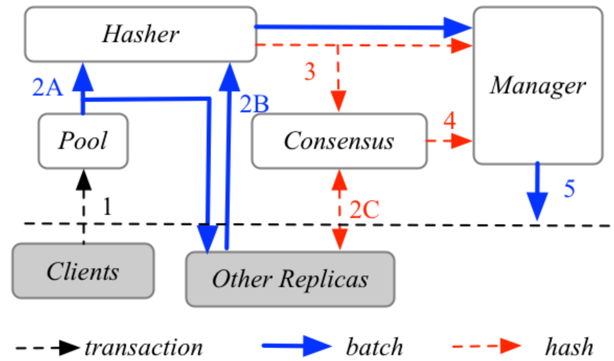


Fig. 54. Architecture of DISPEL. Transactions are collected in the pool and batched (1). Replicas exchange bathes (2A, 2B), store and associate them with their hash digest in the manager (3). Replicas execute consensus protocol over hashes (2C, 3) and transmit the decisions to the manager (4). The manager transmits the batches associated with the decided hashes to the application (5) [280].

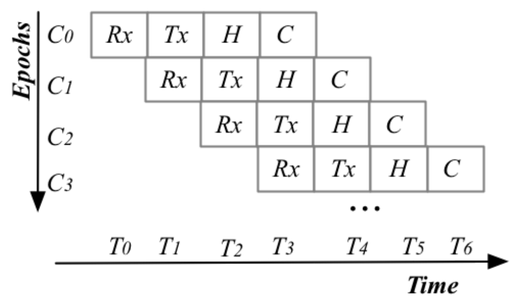


Fig. 55. Principle of a consensus pipeline. A pipeline epoch consists of four phases: network reception (Rx), network transmission (Tx), CPU intensive hash (H) and latency bound consensus (C). As soon as an epoch finishes its first phase (Rx), the replica spawns a new epoch. When the replica executes four epochs concurrently, it leverages all its resources [280].

bottleneck. For the distributed pipelining, it enables the robustness of the distributed system.

In more details, DISPEL works under a partially synchronous model tolerating $f < n/3$ Byzantine replicas, and each replica in DISPEL spawns a new consensus instance based on its available local resources. Fig. 54 shows a high-level architecture and the main steps of one *pipeline epoch* that runs one consensus instance. It mainly consists of creating a batch of commands, exchanging batches with other replicas, and selecting an ordered subset of the batches created by replicas. More specifically, a DISPEL replica continuously listens for client transactions and stores them in its transaction pool. When a replica decides to spawn a new pipeline epoch, it concatenates all transactions in the pool to create a batch, and then broadcasts the batch. In parallel, a dedicated hashing thread computes a checksum of the batch. All DISPET replicas decide independently to spawn a new pipeline epoch. The consensus component exchanges hash to complete the reliable broadcast and execute the subsequent steps. When the replicas decide on a set of hashes, the consensus component transmits its decision to the manager, which gives the decided batches to the pipeline epoch orderer. Based on the above parallel execution, in general, the distributed pipeline contains four distinct phases, as shown in Fig. 55: a network reception phase (Rx), a CPU intensive hash phase (H), a network transmission phase (Tx), and a wait-mostly consensus phase (C). DISPEL executes these phases at the same time from different pipeline epochs, leveraging most resources. Thus, a DISPEL replica spawns a new pipeline epoch before its previous

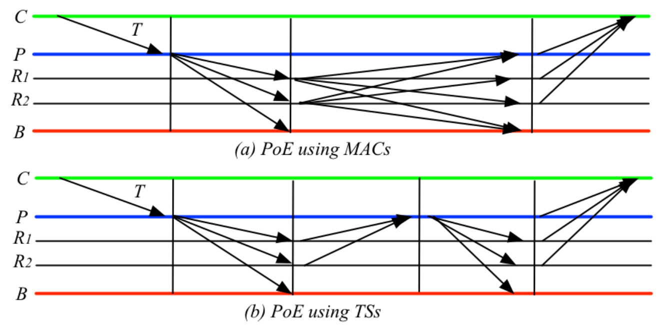


Fig. 56. Normal-case algorithm of PoE: Client c sends its request containing transaction T to the primary P, which proposes this request to all replicas. Although replica B is Byzantine, it fails to affect PoE [282].

epoch terminates. DISPEL starts a new epoch once its resources permit, and each row stands for pipeline epoch and has four phases: Rx, H, Tx, and C. Interested readers can refer to the work [280] for more details.

59) PoE BFT - (architecture): Proof-of-Execution is a BFT consensus protocol that uses speculative execution to reach an agreement among replicas, proposed by Gupta et al. in 2019 [282]. At the core of PoE are out-of-order processing and speculative execution, which allows PoE to execute transactions before a consensus is reached. In general, PoE can achieve resilient agreement in just three linear phases. Compared with PBFT, to make a protocol scalable and resilient, PoE BFT adds four key design elements: non-divergent speculative execution, safe rollbacks and robustness under failures, agnostic signatures and linear communication, and avoid response aggregation. For the non-divergent speculative execution, instead of an all-to-all communication then the execution in PBFT, PoE replicas execute requests after they get *prepared* (the second phases of PBFT), that is, they do not broadcast *Commit* messages. The speculative execution is non-divergent as each replica has a partial guarantee, it has prepared prior to execution. In PBFT, a client may or may not receive a sufficient number of matching responses. For the safe rollbacks and robustness under failures, PoE ensures that if a client receives a *full proof-of-execution*, consisting of responses from a majority of the non-faulty replicas, then such a request persists in time. Otherwise, PoE permits replicas to rollback their state if necessary, which provides a cornerstone for the correctness of PoE. For the agnostic signatures and linear communication, PoE applies a twin-path execution, which allows replicas to either employ message authentication codes (MACs) or threshold signatures (TSs) for signing based on the size of the network. For example, when few replicas are participating in consensus (up to 16), then a single phase of all-to-all communication is inexpensive, and using MACs for such setups can make computations cheap. While for large setups, PoE employs TSs to achieve a linear communication complexity. To avoid response aggregation, different from aggregation schemes (i.e., in SBFT), PoE avoids additional communication between replicas by allowing each replica to respond directly to the client. Fig. 56 shows a normal-case algorithm of PoE.

60) DiemBFT - (feature): DiemBFT (also known as LibraBFT) is a production version of HotStuff to achieve the strong scalability and security properties required by Internet settings, proposed by the LibraBFT teams in 2019 [283]. LibraBFT maintains safety against network asynchrony and even if at any particular configuration epoch, a threshold of participants are Byzantine. It incorporates a round synchronization mechanism that provides to commit despite having

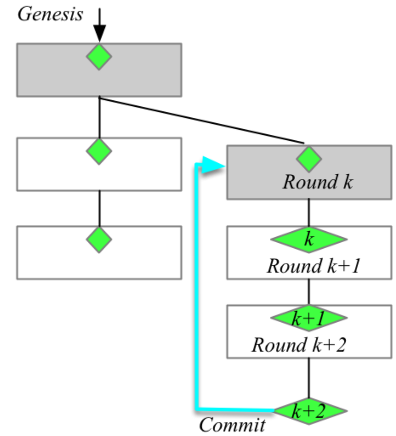


Fig. 57. Proposals (blocks) pending in the block-tree before and after a commit [283].

faulty leaders. It also encapsulates the correct behavior by participants in a “tcb"-able module, allowing it to run within a secure hardware enclave that reduces the attack surface on participants. In addition, LibraBFT can reconfigure itself, by embedding configuration-change commands in a sequence. A new configuration epoch may change everything from the validator set to the protocol itself.

In more details, LibraBFT operates in a pipeline of rounds, in which a leader proposes a new block in each round. Validators send their votes to the leader of the next round, and when a quorum of votes is reached, the leader of the next round forms a quorum certificate (QC) and embeds it in the next proposal. This process continues until three uninterrupted leaders/rounds complete and the tail of a chain has three blocks with consecutive round numbers. Then, the head of the “3-chain" consisting of three consecutive rounds that have formed becomes committed. The entire branch ending with the newly committed block extends the sequence of commits. Fig. 57 shows proposals/blocks pending in the block-tree before and after a commit. It provides a way to chain the blocks. Essentially, there two main components in LibraBFT consensus protocol: a *steady state protocol* and a *PaceMaker (view-change) protocol*. The stable state protocol aims to make progress when the round leader is honest, while the PaceMaker advances the round number either due to the lack of progress or the current round being completed. In general, the LibraBFT consensus protocol has a linear communication complexity per round under synchrony and honest leaders due to the leader-to-all communication pattern and the use of threshold signature scheme, and quadratic communication complexity for synchronization caused by asynchrony or bad leaders due to the all-to-all multicast of timeout messages. However, during the periods of asynchrony, the protocol has *no liveness guarantees*, e.g., the leaders may always suffer from network delays, replicas keep multicasting timeout messages and advancing round numbers, and no block can be certified or committed. Interested readers can refer to the work [283] for more details.

61) Byzantine View Synchronization - (feature): Cogsworth is a Byzantine view synchronization algorithm, that has an optimistically linear communication complexity and constant latency, proposed by Naor et al. in 2019 [96]. The view-based consensus protocols typically have some advantages, e.g., each view has a designated leader that can drive a decision efficiently. A view change occurs as an interplay between a synchronizer, which implements a view synchronization algorithm, and the outer consensus solution. A view synchronization requires to eventually bring all honest nodes to execute the same view for a sufficient long time, for the outer consensus protocol to be able to drive progress. However, existing approaches for Byzantine view synchronization incur quadratic communication (e.g., in n, the number of replicas). And a cascade of $O(n)$ view changes may thus result in an $O(n^3)$ communication complexity. Cogsworth is a leader-based view synchronization algorithm to reduce this communication complexity. Essentially, Cogsworth utilizes views that have an honest leader to relay messages, instead of broadcasting them. When a node wishes to advance a view, it sends the message to the leader of the view, instead of to all nodes. If the leader is honest, it will gather messages from nodes and multicast them using a threshold signature to other nodes, incurring only a linear communication cost. Cogsworth implements additional mechanisms to advance views despite faulty leaders. More preciously, if a leader of a view (e.g., v) is Byzantine, it might not help as a relay. In this case, the nodes time out and then try to enlist the leaders of subsequent views, one by one, up to view $v + f + 1$, to help with relaying. At least one of $f + 1$ leaders is honest, thus one of them will successfully relay the aggregate.

In general, the latency and communication complexity of Cogsworth depend on the number of actual failures and their types. For example, in the best case, the latency is constant and the communication is linear. When there exist t benign failures, the communication is linear and, in the worst case $O(t \cdot n)$, the latency also is expected to be constant (e.g., $O(t \cdot \delta)$ in the worst-case). And Byzantine failures do not change the latency, but they can drive the communication to an expected $O(n^2)$ complexity, and in the worst-case up to $O(t \cdot n^2)$.

Another work on view-change protocol, which targets an asynchrony setting, is proposed by Gelashvili in 2021 [113]. The proposed work is a BFT SMR protocol with the feature of "prepared" when the network goes bad. Essentially, when the network behaves well and the consensus makes progress, it runs DiemBFT [283] (an implementation of HotStuff [284]) taking its linear communication complexity. However, when there is a problem and the majority of nodes cannot reach the leader, it proactively runs an asynchronous view-change protocol that makes progress even under a strongest network adversary. The proposed asynchronous view-change protocol, also called asynchronous fallback [285], has quadratic communication complexity and always makes progress even under asynchrony. The asynchronous fallback protocol can replace the Pacemaker protocol in DiemBFT to obtain a BFT protocol that has linear communication cost for synchronous paths, quadratic cost for asynchronous paths, and is always live.

62) HotStuff - (feature): HotStuff is a leader-based Byzantine fault-tolerant replication protocol for a partially synchronous model to achieve responsiveness, proposed by Yin et al. in 2019 [284]. It ensures that once the network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of the actual network delay, a property of responsiveness. And this network delay is with a communication complexity that is linear in the number of replicas. The simplicity design of HotStuff enables it to be further pipelined and simplified into a practical and concise protocol for building large-scale replication services. In general, a stable leader can drive a consensus decision in just two rounds of message exchanges: the first phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of $(n-f)$ votes, and the second phase guarantees that the next leader can convince replicas to vote for a safe proposal. However, for a new leader-based protocol, with the view-change, it is far from simple than the one based on the two-phase paradigm, and is bug prone [80], and incurs a significant communication penalty for even moderate system sizes. It requires the new leader to relay information from $(n-f)$ replicas, each reporting its own highest known QC. The total number of authenticators (e.g., digital signatures or message authentication codes) transmitted is pretty high, e.g., (n^4) in PBFT, and (n^3) in threshold signature variants, where n is the number of participating replicas. HotStuff revolves this around a three-phase core, allowing a new leader to simply pick the highest QC it knows of. It introduces a second phase that allows replicas to "change their mind" after voting in the phase, without requiring a leader proof at all. This can alleviate the above complexity, and at the same time considerably simplify the leader replacement protocol. Besides, by having almost containerized all the phases, it is very easy to pipeline HotStuff and to frequently rotate leaders.

In more details, responsiveness requires that a non-faulty leader in a leader-based BFT protocol, once designated, can drive the protocol to consensus in a time depending only on the actual network delay, independent of any known upper bound assumption on message transmission delays [257] [255]. In HotStuff, *optimistic responsiveness* requires responsiveness only in beneficial (e.g., common case) circumstances, e.g., after Global Stabilization Time (GST) is reached. The crux of the difficulty is that there may exist an honest replica that has a highest QC, but the leader does not know about it. It can build scenarios where this prevents progress of the whole system. The HotStuff can achieve two main properties: linear view change, and optimistic responsiveness. For the linear view change, after GST, any correct designed leader sends only $O(n)$ authenticators to drive a consensus decision, including the case where a leader is replaced. Thus, in the worst case of cascading leader failures, its communication costs to reach a consensus after GST is $O(n^2)$. For the optimistic responsiveness, after GST, any correct designed leader needs to wait just for the first $(n - f)$ responses to guarantee that it can create a proposal that will make progress, including the case where a leader is replaced. It is noticeable that the cost for a new leader to drive the protocol to consensus is no greater than that for the current leader. And, HotStuff supports frequent succession of leaders, which has been argued being useful in a blockchain context to ensure the chain quality [286]. In general, HotStuff is via adding another phase to each view, a small price to latency in return for considerably simplifying the leader replacement protocol. And this exchange incurs only the actual network delay, which is typically far smaller than Δ in practice. Also, the throughput is not affected due to some efficient pipeline schemes. Besides, in HotStuff, safety is specified via voting and commit rules over graphs of nodes, and liveness are encapsulated within a *Placemaker*, clearly separated from the mechanisms needed for safety.

Regarding the response latency, in HotStuff, a leader of each view proposes a block, and a block is decided after three blocks of consecutive views are certified. Each view consists of two rounds of communication. Thus, the latency is $[6\delta, 8\delta]$, and the average latency is 7δ. Interested readers can refer to the work [284].

63)Sync HotStuff - (feature): Sync HotStuff is a simple and intuitive leader-based synchronous BFT solution, proposed by Abraham et al. in 2020 [261]. It can achieve consensus within a latency of 2Δ in a steady-state, where Δ is a synchronous message delay upper bound. Sync HotStuff can ensure safety in a weaker synchronous model in which the synchrony assumption does not have to be held for all replicas all the time. Even that, it has optimistic responsiveness, e.g., it advances at network speed when less than one-quarter of replicas are not responding. By combining solutions from some practical partially synchronous BFT protocols, Sync HotStuff has a two-phase leader-based structure, and has been full prototypes under the standard synchrony assumption. In general, synchronous BFT protocol has the advantage of tolerating up to one-half Byzantine faults, while asynchronous or partially synchronous protocols tolerate only one-third. However, most synchronous protocols have an issue of lock-step execution, where replicas must start and end each round at the same time, and the issues of impracticalness and unsafeness (e.g., subjecting to network attacks on timing assumption). In Sync HotStuff, each replica can commit after waiting for the maximum round-trip delay unless it hears by that time an equivocating proposal signed by the leader. Sync HotStuff achieves several desirable features: 1) it tolerates up to one-half Byzantine replicas; 2) it does not require lock-step execution in a steady-state; 3) it can handle a weaker and more realistic synchrony model; 4) it is prototyped and shown to offer practical performance. And there indeed have some use cases, e.g., in the single-datacenter replicated services and consortium blockchain applications.

In more details, Sync HotStuff provides several advantages compared with other synchronous BFT. The first one is its near-optimal latency. Sync HotStuff is with a simple steady-state protocol, in which replicas are just to wait for a single maximum round-trip delay sufficing to commit, and it does not have to be executed in a lock-step fashion, As long as replicas receive enough messages of the previous step, they can move to the next step without waiting for the conservative synchrony delay bound. This gives a latency of $2\Delta + O(\delta)$ in a steady-state, where Δ denotes the known bound assumed on the maximal network transmission delay and δ denotes the actual network delay, which is much smaller than Δ. The intuition of 2Δ is that replicas can be out-of-sync by Δ, so one Δ is needed for lagging replicas to catch up, and another Δ is needed for messages to be delivered. Even though $O(\delta)$ latency is impossible to guarantee under more than one-third of faults, it can be achieved optimistically. The second one is a practical throughput. This is achieved by moving the synchronous waiting steps off the critical path, and the only step in a steady state that requires waiting for a conservative $O(\Delta)$ time is to check for a leader equivocation before committing, and it can be performed concurrently with the main logic. So replicas start working on the next block without waiting for the previous blocks to be committed. Even though there are other synchronous waiting steps in a view-change protocol, it occurs infrequently. The third one is the safety despite some sluggish honest replicas. The sluggish means it allows a set of honest replicas to violate the message delay bound Δ at any point in time. Message sent by or sent to a sluggish replica may take an arbitrarily long time until replicas are prompt (aka. behave honestly) again. The authors prove that Sync HotStuff ensures safety as long as the combined number of sluggish plus Byzantine faults is less than one-half, which means at any time, a majority of replicas must be honest and prompt. The evaluation on the proposed prototype shows that Sync HotStuff can achieve a throughput of over 280 Kops/sec under typical network performance.

One consideration on Sync HotStuff is that, in each view, at least one block must be decided with a synchronous latency to switch to the responsive path, which is not responsive in essence. And, Sync HotStuff cannot decide a block synchronously in each view once it switches to the responsive mode. Thus, Sync HotStuff cannot decide blocks in a view if the number of actual faults is f but the faulty parties first behave honestly until the protocol switches to the responsive mode and crash after that.

64) Fast-HotStuff - (feature): Fast-HotStuff is a two-round BFT protocol, based on the HotStuff protocol, to achieve responsiveness and an efficient view-change process that is comparable to linear view-change in terms of performance, proposed by Jalalzai et al. in 2020 [287]. Compared to the three-round HotStuff, Fast-HotStuff, by re-establishing a two-round consensus, claims lower latency and robustness against performance attacks that HotStuff is susceptible to. In general, classic BFT protocols do not allow fork formation, while forking in HotStuff may occur, and results in lower throughput and higher latency. For example, a malicious primary can use an old QC (Quorum Certificate: containing the parent block hash and $(n - f)$ votes from replicas for the parent block) instead of the latest one to construct a new protocol. This forking attack in a pipelined HotStuff is possible, and this is valid if the QC is no more than two views older than the latest one according to the HotStuff's voting rule. And thus, the previous not committed block will be forked. To circumvent these issues, Fast-HotStuff re-shapes HotStuff to reserve and achieve some innovative properties: low latency, responsiveness, scalability, and robustness against fork attacks. The first three properties are almost similar to the properties of HotStuff. By reducing a three-round communication latency to two rounds, the block in Fast-HotStuff can be committed with around 30% less delay. For robustness against fork attacks, a primary in Fast-HotStuff has to provide a *proof* that it has included the latest QC (a pointer to its parent block) in the proposed block, which is unlike the scenario of HotStuff. By providing the proof of the latest/highest QC, a Byzantine primary cannot perform a forking attack. This is because the Byzantine primary cannot use an older QC due to the requirement of the proof of the latest QC included in the block. Also, this proof helps a replica to achieve consensus within two rounds. It provides the guarantee that if a replica committed a block at the end of the second round, eventually all other replicas will commit the same block at the same height. Thus, a replica does not need to wait for a maximum network delay to make sure that other replicas have also committed the same block.

More specifically, Fast-HotStuff operates in a series of views with monotonically increasing view numbers, and each view number is mapped to a unique dedicated primary known to all replicas. The optimistic commit/execution to achieve low latency is valid when either one of the two conditions is met: 1) the block proposed by the primary is built by using QC that is the highQC (a kind of QC) held by the majority of honest replicas, or 2) by a higher than the highQC being held by a majority of replicas. Thus, the primary has to incorporate the proof of the highQC in every block it proposes, which can be verified by every replica. Interested readers can read the work [287] for more details.

65) Pompe - (middleware): Pompe is a BFT SMR protocol that is guaranteed to order commands in a way that respects a natural extension of linearizability, proposed by Zhang et al. in 2020 [288]. The authors introduce a Byzantine ordered consensus, as a primitive, to augment the correctness specification of BFT SMR to include a specific guarantee on the total orders, and an architecture for BFT SMR that, by factoring out ordering from the consensus, can enforce these guarantees and prevent Byzantine nodes from controlling ordering decisions (a Byzantine oligarchy). According to the authors' observation, the traditional specification of correctness for SMR is intrinsically incapable of characterizing what makes a total order "right" or "wrong". The authors intended to provide a primitive to

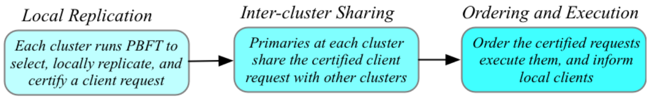


Fig. 58. Steps in a round of the GEOBFT protocol [289].

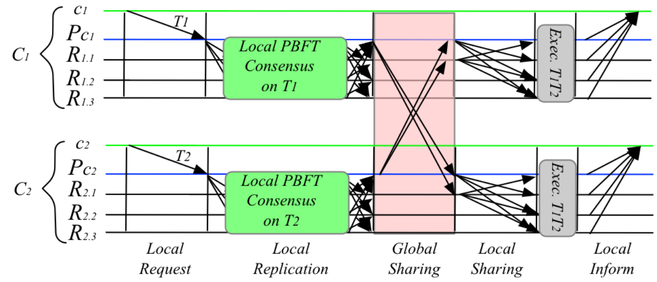


Fig. 59. Representation of the normal-case algorithm of GeoBFT running on two clusters. Client c_i, $i \in \{1, 2\}$, requests transactions T_i from their local cluster C_i. The primary $P_{c_i} \in C_i$ replicates this transaction to all local replicas using PBFT. At the end of local replication, the primary can produce a cluster certificate for T_i. These are shared with other clusters via inter-cluster communication, after which all replicas in all clusters can execute T_i and C_i can inform c_i [289].

augment the correctness specification of the BFT SMR, which allows the nodes that implement a replicated state machine to associate an ordering indicator to the commands they ultimately agree upon. Through these indicators, nodes can express how they would like these commands to be ordered with respect to another. The correctness conditions for the Byzantine ordered consensus specify, given a set of input <ordering indicator, command> pairs, the set of allowable total orders for these commands. Besides, the authors also introduce a general architecture for BFT protocols that factors ordering out of consensus, which cleanly separates the process of establishing the relative order of commands from the consensus steps necessary to add those ordered commands to the ledger.

66) GeoBFT - (middleware): GeoBFT is a geo-scale Byzantine fault-tolerant consensus protocol to achieve scalability by using a topological-aware grouping for replicas in local clusters, proposed by Gupta et al. in 2020 [289]. In general, it is desirable that the underlying consensus protocol must distinguish between local and global communication. However, existing Byzantine consensus protocols are not designed to handle geo-scale deployments in which many replicas spread across a geographically large area. GeoBFT can solve this issue by introducing the parallelization of consensus at a local level, and by minimizing the communication between clusters. To get a geo-scale aware consensus protocol, it needs to be aware of the network topology, which can be achieved by clustering replicas in a region together and favoring communication within such clusters over a global inter-cluster communication. Also, a geo-scale consensus needs to be decentralized, which means no single replica or cluster should be responsible for coordinating all consensus decisions. This is because a centralized design limits the throughput to outgoing bandwidth and the latency of this single replica or cluster.

In more details, GeoBFT group replicas in a region into clusters, and each cluster makes consensus decisions independently. These consensus decisions are then shared via an optimistic low-cost communication protocol with the other clusters. By doing so, it assures that all replicas in all clusters are able to learn the same sequence of consensus decisions. For example, for two clusters C_1 and C_2 and each with n replicas, the optimistic communication protocol requires only $n/3$ messages to be sent from C_1 to C_2 when C_1 needs to share its local consensus decision with C_2. By doing so, it can help the throughput and scalability in a geo-scale deployment. In general, GeoBFT operates in rounds, and in each round, every cluster will be able to propose a single client request for execution. From a high level perspective, each round consists of three steps, as shown in Fig. 58, *local replication*, *global sharing*, and *ordering and execution*. Fig. 59 shows a representation of the normal-case algorithm of GeoBFT running on two clusters, following these three steps. For the local replication step, each cluster chooses a single transaction of its local client, and each cluster locally replicates its chosen transaction in a BFT manner, e.g, using PBFT. At the end of local replication, the adopted BFT protocol guarantees that each non-faulty replica can prove successful local replication via a *commit certificate*. For the global sharing step, each cluster shares locally replicated transaction along with its commit certificate with all other clusters. This step utilizes an optimistic global sharing protocol to minimize inter-cluster communication, in which has a global phase to exchange locally replicated transactions, followed by a local phase in which clusters distribute any received transactions locally among all local replicas. In case of failures, the global sharing protocol utilizes a remote view-change protocol. For the ordering and execution step, after global sharing, each replica in each cluster can deterministically order all these transactions and proceed with its execution. Once executed, the replicas in each cluster inform only local clients of the outcome of the execution of their transactions.

The authors provide proofs on safety and liveness of GeoBFT. For the safety, it achieves a unique sequence of consensus decisions among all replicas and ensures that clients can reliably detect when their transactions are executed. For the liveness, whenever the network provides reliable communication, GeoBFT continues successful operation. Also, the authors deployed GeoBFT on their ResilientBD fabric [290] to evaluate the performance.

Besides the GeoBFT, there exist several other literatures for geo-replicated state machines for wide area networks, e.g., Steward proposed by Amir et. al in 2008 [291], WHEAT proposed by Sousa et al. in 2015 [292], and AWARE proposed by Berger et al. in 2020 [293]. Steward is a hierarchical BFT protocol for geographically dispersed multi-side systems, which employs a hybrid algorithm that runs a BFT agreement within each site, and a lightweight, crash fault-tolerant protocol across sites. WHEAT is a variant of BFT-SMART protocol that is optimized for geo-replicated environments, which employs a voting assignment scheme that, by using few additional spare replicas, enables the system to make progress without needing to access a majority of replicas. Its main innovation is the ability to decrease the client latency by, counter-intuitively, adding more replicas to the system. AWARE is an adaptive wide-area replication system for a fast and resilient Byzantine consensus, which employs an automated and dynamic voting-weight tuning and leader positioning scheme to support the emergence of fast quorums in the system. It utilizes a reliable self-monitoring process and provides a prediction model seeking to minimize the system's consensus latency. Interested readers on geo-replicated state machines can refer to these corresponding papers.

67) FNF-BFT - (architecture): FNF-BFT (Fast'N'Fair-BFT) is a parallel-leader BFT protocol for a partially synchronous model with theoretical performance bounds during synchrony, proposed by Avarikioti et al. in 2020 [294]. By allowing all replicas to act as leaders and propose requests independently, it parallelizes the execution of requests, which circumvents the common single-leader bottleneck and achieves a significant performance increase over the sequential-leader systems. In general, the communication complexity of FNF-BFT is linear in the number of replicas during the synchrony, and it provides guarantees in stable network conditions under truly Byzantine attacks. FNF-BFT uses a better Byzantine-resilient performance metric, which encapsulates the ratio between the best-case and worst-case throughput of a BFT protocol, to evaluate a BFT's performance. And FNT-BFT can achieve Byzantine-resilient performance with a ratio of $16/27$ while maintaining both safety and liveness. More specifically, FNF-BFT, as a multiple-leader BFT protocol, achieves the following three properties under a stable network condition: optimistic performance, Byzantine-resilient performance, and efficiency. The optimistic performance means, after GST, the best-case throughput is $\Omega(n)$ times higher than the throughput of sequential-leader protocols; the Byzantine-resilient performance means that, after GST, the worst-case throughput of the system is at least a constant fraction of its best-case throughput; and the efficiency means that, after GST, the amortized authenticator complexity of reaching consensus is $\Theta(n)$. To achieve these three properties, FNF-BFT utilizes two key mechanisms. The first mechanism is to enable all replicas to continuously act as leaders in parallel to share the load of client's requests; the other one is that FNF-BFT does not replace leaders upon failure but configure each leader's load based on the leader's past performance. By combining these two mechanisms, FNF-BFT can guarantee a *fair* distribution of requests according to each replica's capacity, which in turn results in *fast* processing of requests. Besides, the authors also implement an FNF-BFT prototype and compare it with HotStuff's throughput, it shows that, as replicas increase, the FNF-BFT protocol achieves a significant improvement on scalability and exhibits faster average performance.

68) Twins - (architecture): Twins is an approach for testing BFT systems, which requires only a thin network wrapper that delivers messages to/from both twins, proposed by Bano et al. in 2020 [295]. The key idea of Twins is that it emulates Byzantine behavior by running two (or generally up to k) instances of a node with the same identity. Each of the two instances or twins runs unmodified, correct code. To the tested system, the twins appear indistinguishable from a single node behaving in a 'questionable' manner. The current version of Twins can generate several (not all) Byzantine behaviors, including equivocation, double voting, and losing internal state, while forgoing some other behaviors that are trivially rejected by honest nodes, such as producing semantically invalid messages. By this design, Twins can systematically generate Byzantine attack scenarios at scale, execute them in a controlled manner, and check for desired protocol properties. In Twins, instead of coding incorrect behaviors, it runs faulty nodes in two parallel universally, and both instances have the same credentials/signing-keys and run autonomously. For instance, both instances can send messages in a same protocol round, however, these messages carry conflicting information. And to the rest of the system, this twins behavior will appear indistinguishable from an equivocating behavior by a single node. Also, the authors performed some tests on existing protocols, which are broken within fewer than a dozen protocol steps, and it is realistic for the Twin approach to expose problems.

More specifically, Twins is a "white glove" approach. It is neither a "black-box", since it does modify the internal behavior of the tested system, nor is it a "white-box", as it does not open internal code modules. In general, Twins minutely interacts with existing codes to control message delivery and schedule various coarse-steps such as protocol rounds. It can be deployed in a real system as it uses

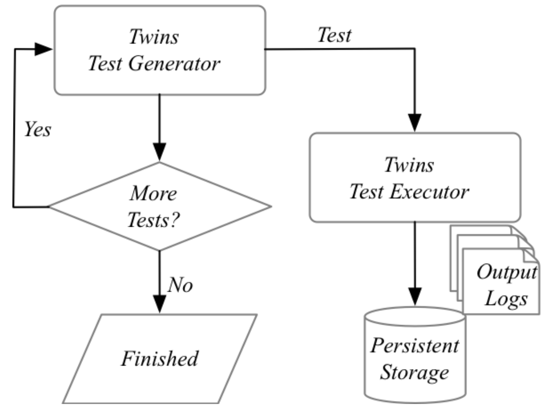


Fig. 60. Twins high-level design [295].

the existing correct node code. Also, Twins can be implemented by thinly wrapping twin nodes with a network-scheduler acting as an adversary, easily keeping up with an evolving software project. Implementing Twins typically consists of two principal parts: *test executor* and *test generator*. Fig. 60 shows a high-level design. The test executor deploys a network configuration where some nodes have twins, and it hides twins behind a thin multiplexing wrapper; while the test generator enumerates scenarios by varying the number of nodes and the message delivery schedule, then feeding the scenarios to the test executor. For more details, interested readers can refer to the work [295].

An almost similar work is presented at [296], called *Gemini*, by the same authors of Twins. It follows almost all descriptions and scenarios of Twins. However, it might have different terminologies on some functional parts. For example, Twins has two principle parts named *test executor* and *test generator*, while, in Gemini, these two principle parts are named *scenario executor* and *scenario generator* only with different names.

69) GRANDPA - (feature): GRANDPA is a protocol to solve the Byzantine finality gadget problem, proposed by Stewart and Kokoris-Kogia in 2020 [297]. In general, classic BFT protocols forfeit liveness under an asynchronous assumption to preserve safety, while most deployed blockchain protocols forfeit safety in order to remain live. To achieve the best of both, an abstraction would be required in the form of *finality gadget*, which allows for transactions to always optimistically commit but informs the clients that these transactions might be unsafe. For example, a blockchain can execute transactions optimistically and only commit them after they have been sufficient and provably audited. The authors propose a formal model on the finality gadget abstraction, and proved that it is impossible to solve it deterministically in a full asynchrony environment. The authors also present a partially synchronous protocol that is currently used to secure some major blockchains, so that the protocol designer can decouple safety and liveness to speed up recovery from failures. For a blockchain protocol, it has several advantages to separate the liveness of the consensus protocol from the finality of blocks. 1) Consensus is not tied to the liveness of the chain, which can have optimistic execution, so that the chain can grow before it is certain that blocks are valid. Later on, the chain can finalize blocks when it sures they are correct, e.g., all verification information is available. 2) The protocol can make some (even unsafe) progress when the network is unstable, which enables fast recovery progress when the network heals. And similarly, the chain can also make some progresses even when the finalization is slow. 3) A finality gadget can be deployed gradually and light clients can choose to consult it or follow the longest chain rule and ignore it, enabling the heterogeneity of light clients.

More specifically, the authors show it is impossible to satisfy some features of finality gadget with a deterministic asynchronous protocol, and introduce the GRANDPA finality gadget that works in a partially synchronous network model. GRANDPA works in rounds, each round has a set of $3f+1$ eligible voters to tolerate up to f faulty voters. Each round consists of a double-echo protocol after which every party waits to detect whether it can finalize a block in this round. In this case, the block does not need to be an immediate ancestor of the last finalized block, it might be far ahead from the last finalized block. If the round is unsuccessful, the parties simply move on to the next round with a new primary. When a good primary is selected, the oracle is consistent (e.g., returning the same value to all honest parties), and the network is synchrony (after GST), a new block will be finalized and it will transitively finalize all its ancestors. For more details on the implementations and proofs, interested readers can refer to the work [297].

70)SFT - (feature): Strengthened Fault Tolerance (SFT) is a BFT SMR under a partial synchrony that provides gradually increased resilience guarantees during an optimistic period (e.g., with a synchrony network and a small size of Byzantine faults), proposed by Xiang et al. in 2021 [298]. The committed blocks can tolerate more than one-third (up to two-thirds) corruptions even after the optimistic period. This means the safety assurance of a decision can be made to improve over time, like Nakamoto consensus, and the blockchain can obtain resilience against a higher number of Byzantine failures as it gathers additional confirmations (e.g., votes for the block). Compared with a similar solution of Flexible BFT (FBFT) [275] (requiring a quadratic message complexity), SBFT maintains a linear message complexity over BFT SMR protocols and requires only a marginal bookkeeping overhead. The term *strengthened* in SFT reflects the fact that stronger resilience guarantees can be obtained given that the conditions are optimistic, e.g., when the network is synchronous and the number of Byzantine faults is small during the optimistic period. Blocks committed with a higher resilience mean much safer even if the number of Byzantine faults later exceeds the one-third threshold. As the rule of *k*-deep in Nakamoto consensus, with SFT, the clients can choose to wait longer for valuable blocks to obtain higher resilience, trading off safety with latency. Also, SFT induces only a marginal bookkeeping overhead and retains a *linear message complexity*, while the adaptability of FBFT incurs a quadratic message complexity overhead. Besides, the authors implement SFT over DiemBFT (or called LibraBFT) [283], called SFT-DiemBFT, whose implementation adds only moderate latency increase to obtain higher assurance guarantees (up to two-thirds) under various conditions. Interested readers can refer to the work [298] for more details.

V. ESSENTIAL COMPONENTS OF BFT

BFT replication protocols make it possible to design systems that are resilient against arbitrary faults, which can be potentially applied to many crucial use cases, such as blockchain, datacenters, and various decentralized applications. In general, the guarantees on fault-tolerance come at some cost of the complexity (in both communication and computation), which inherently makes them hard to incorporate in practice BFT replication systems. We discuss the state-of-the-art BFT protocols in Section IV, and each protocol was its advantages and disadvantages. Most of these solutions are designed to improve performance, robustness, availability, and resource efficiency under various assumptions. It is necessary to demystify the basic building blocks of BFT protocols from a high-level perspective. There are several literatures discussing this kind of demystification. For example, Correia et al. [16] analyze different BFT consensus algorithm from the theoretical level; Berger and Reiser [17] present work on improving scalability when applying Byzantine consensus to blockchains and distributed ledgers; Platania et al. [299] compare BFT protocols based on the role of clients; Dister [6] presents a generic BFT state-machine replication from a systems perspective. Based on these works, we present some essential components to the construction of BFT replication protocols, which can help system designers design their own application-specific BFT consensus protocols. Roughly speaking, these components include clients, an agreement protocol, an execution stage, a checkpointing scheme, and a replica recovery scheme. Note that it is not necessary for a BFT consensus protocol to include all components, and some protocols may combine two or more components together. Also, they may relate to other techniques, such as a checkpointing scheme that relates to garbage collection in SBFT or replica recovery scheme that relates to the view change in PBFT.

Most existing BFT protocols consist of three clear tasks, especially for the cases of blockchain applications, namely *client handling*, *agreement*, and *execution*. According to the classification of Platania et al. [299], these three procedures are the “server"-side tasks, in which clients only have the right to access the replicated system (e.g., submitting requests) without actively participating in consensus procedures. Roughly speaking, in a client handling process, replicas need to receive and disseminate requests from clients and, once the request is processed and a decision is made, return the responses to clients. This task typically involves a significant amount of communication and computational overhead (e.g., processing authenticated messages) between clients and replicas. And the overhead of this task generally relies on the number of active clients and the workload of requests. Thus, preventing Sybil attacks and DoS attacks are important in this task. In the agreement process, replicas are required to reach a consensus on the decisions of requests, and typically, this task involves only the communication among replicas. The communication complexity of this process depends on the size of replicas in the system. And, due to the existence of Byzantine replicas, it may require some period of synchronization to successfully transmit messages among non-faulty replicas. In the execution process, replicas, once agreed on the requests, are required to process these requests based on their current state. This process involves updating the local state information of replicas. The complexity and overhead of this process typically depend on the specific applications, e.g., appending blocks to the blockchain. The organization and dependency of these three tasks [6] result in different system architectures, e.g., a one-tier architecture (each replica performing client handling, agreement, and execution all together), a two-tier architecture (each replica performing client handling and agreement together, and performing execution separately), and a three-tier architecture (each replica performing three tasks separately).

Most existing classic BFT protocols include the above three tasks in order to reach an agreement. Besides that, there are some new architectures, which follow different procedures, such as the *execute-verify* principle proposed by Kapritsos et al. [300]. The execute-verify principle makes it possible that the agreement process and execution process are handled in a different order (e.g., out-of-order execution). Compared with “agreement - execution" in traditional BFT protocols, the execute-verify principle allows the replicated system to perform the applications outputs and replicated state updates separately, which helps them to establish an order on the requests later by utilizing consensus protocols. By doing so, responsiveness to clients' requests can be improved. Also, this principle typically targets permissioned blockchains, which require access control on replicas to participate in the consensus process. In the following sections, we discuss some generic components according to the classification of Dister [6].

A. Client

The component of the client provides an interface to access the replicated systems. Though we can literally consider them as users, they are not actual users who issue requests to the replicated systems. Besides simply relaying requests and replies, the clients typically take several roles to connect the user processes and the replicated systems: communication with replicas, result verification, handling of replica inconsistencies, and provision of optimization hints.

One of the primary tasks of clients is to communicate with replicas, and this communication is typically based on the "client-server model", e.g., following the requests and replies messages over the network. Also, the client may take some responsibility to handle exceptions. For example, if the network is not reliable, it may involve the tasks of re-transmission of messages, or re-establishment of broken connections. By taking these responsibilities, clients can ensure that the requests of clients will eventually be executed even they may connect to faulty replicas (e.g., as agents of users). There are two major approaches for clients relaying the requests. One approach is the request broadcast, in which clients directly submit the corresponding request to all replicas of the replicated system. This approach can ensure that non-faulty replicas can learn the requests and thus detect the situations in which the leader tries to suppress a client's requests. Another approach is the use of a contact replica, e.g., the leader in leader-based BFT protocols, in which a client only submits the requests to a single contract replica. In general, this approach is optimistic and can work well in fault-free cases. However, this approach typically requires an additional setup on timeout to handle the case of contact replica failure. When timeout is reached, the client can choose to multicast these requests. For the task of communication with replicas, there exist some respective works to describe its functionality, e.g., Troxy [254] and SBFT [65].

Due to the existence of Byzantine replicas, clients must have some ability to handle the abnormal replies; thus the client can verify replies and handle the inconsistencies from replicas. In a Byzantine setting, a client typically is required to wait for replies from different replicas and then compare those results before handling them to users. The number of matching replies depends on the consensus protocol running on a replicated systems. Typically, it will require at least $f+1$ matching replies from different replicas, where f is the maximal number of faulty replicas a consensus protocol can tolerate. In case of the equivocation of replicas, e.g., if a faulty leader makes conflicting proposals to correct follower replicas that diverge, clients must have the ability to detect them. For example, Zyzzyva includes the order information into the executed requests then replies to the clients with that information, which allows the clients to have the ability to detect inconsistency of running results. Also, on the detection of replica inconsistencies, clients must provide some solutions to resolve. For example, in Abstract protocol [82], it relies on an active switching protocol, which allows the system to make a transition from one state (e.g., view) to another by relying on the execution history information. Clients are responsible to collect and distribute this history information to help the replicas make decisions. And in general, a single active correct client is sufficient to complete this switching process. Besides, there are some protocols relying on clients to provide the optimization. For example, the Archer [301] exploits clients to assist in dynamically configured geo-replicated systems to improve the responsiveness and reduce the end-to-end response time. In Archer, clients exploit the location of the leader to offer the best performance for the current workload conditions.

B. Agreement

The primary goal of the agreement stage is to establish a total order on client requests globally and stably, which further can be used for the execution stage to execute these ordered requests among replicas of the replicated systems. Simply, this stage is used to get agreement on the final executed requests. Assuming the requests submitted from clients are correct, then this stage is about establishing a "timeline" (e.g., timestamp) on different requests from the same client or different clients. According to the definition on an unstable timeline (originating from a single source, maybe Byzantine replicas) and stable timeline (originating from a group of nodes, guaranteed correctness) in the work [6], the agreement typically adopts three building blocks: a nondeterministic merge, an agreed merge, and a deterministic merge.

Simply, a nondeterministic merge takes a set of unstable timelines and merges them into a combined unstable timeline; an agreed merge transforms s set of unstable timelines into a stable timeline, and a deterministic merge combines a set of stable timelines as input and outputs a single stable timeline. In general, the nondeterministic merge happens locally, so that there is no global agreement on this merge and each replica may still have different timelines on requests. The agreed merge is used to guarantee that all correct replicas can have a consistent view of the interleaving and unstable local inputs or requests from clients, and this building block typically requires multiple phases of network communication. The deterministic merge is guaranteed by the existence of a deterministic algorithm that should be known to all correct replicas. Thus, the agreement stage can be considered as a chain of merge operations, and different protocols may take different merge steps to get an agreement. For example, PBFT [4] takes only one step via agreed merge over all requests by primary; Prime [188] takes a two-step process: first performs non-deterministic merges globally then performs agreed merge globally; and COP [229] also takes a two-step process: first, perform agreed-to merge locally, then perform the deterministic merge globally. Here the terms "locally" and "globally" mean that replicas consider a local set of requests and all requests as global, respectively.

When clients submit requests to the replicated systems, it does not mean that the consensus must perform the agreement stage on client requests. In general, the inputs to an agreement stage can be in three forms: full requests, request hashes, and progress vectors. Full requests as inputs are the straightforward way, in which each replica operates on the client requests directly. By doing so, multiple requests may be batched into one consensus instance to the replicated systems [302]. When replicas directly exchange full requests during agreement, it may significantly increase message sizes, which causes some performance overhead. Most BFT systems perform agreement on request hashes proposed by Castro et al. in 2002 [15], which is more efficient. However, this approach requires additional processing to make a decision on client requests. For example, replicas typically do not approve a leader's request proposal solely based on some request hashes, and they must wait until all requests have been gathered from different sources before actually processing these requests. A faulty client may send a different version of the requests to the leader and other replicas separately, which causes an inconsistency on client requests. If applying request hashes only, the correct replicas may not have the ability to validate the proposal from a correct leader. Although replicas can retroactively fetch the full version of the proposed requests from the leader, it involves a more complex fault-handling mechanism and takes valuable time to make agreement progress. This will make it worse for batch processing. For instance, a missed or diverged request from a faulty client may prevent an entire batch of requests from being confirmed, which definitely affects the responsiveness of correct requests from correct clients. In the approach of progress vectors as an input, consensus protocol does not operate directly on actual requests or their hashes, instead, they insert the requests into replica-local timelines, and the consensus protocol operates directly on these timelines. Since these replicas timelines are reliably distributed, they can be considered as a progress vector which contains basic information on the sequence of requests. Replicas can easily compute the progress vectors in a deterministic way, and the agreement stage can later deterministically translate back to the client requests. In this approach, there is no relation between the size of consensus messages and the number of requests from a client in one round of consensus. Prime [188] utilizes this approach. Besides, Dister [6] gives the use of application outputs as the input to the agreement stage. This mainly has the advantage of the *execute-verify* principle [300], in which the consensus protocol is performed on the executed application outputs.

C. Execution

The execution stage is used to process client requests and produce the corresponding replies to clients, which is application-specific. By separating the execution stage from (application-independent) agreement stage, the overall BFT protocol can not only be more modularized in design (e.g., taking advantage of the pipeline), but also can prevent it from becoming a performance bottleneck. In general, by doing so, it provides two advantages: separating execution replicas and relaxing execution order.

Separating execution replicas can provide some benefits. For example, as stated by Yin et al. [176], this would separate responsibilities for different operations and perform these operations on dedicated processes, which can help to speed up the whole BFT consensus. Their work is achieved by performing the agreement stage and execution stage separately, which literally offers a privacy firewall and isolation entity to prevent the actual execution (e.g., modifying internal states) from faulty replicas. There are some other works adopting this methodology, e.g., UpRight [198], VM-FIT [303], Spare [304] and ZZ [215]. Also, separating execution replicas can significantly simplify the parallelization of the agreement process, such as performance optimization works of COP [229] and Omada [305]. Another key benefit is that it can reduce the number of application instances required for fault tolerance. As Yin et al. [176] pointed out, it only requires $2f + 1$ replicas for the execution of requests to tolerate up to f faulty replicas, instead of at least $3f + 1$ replicas for the agreement of requests. If the replicated systems are application targeted, in which the application is the most complex part, the lower number of execution replicas can significantly reduce the system overheads.

However, separating execution replicas is not a one-shot solution, and it leads to some special requirements and obstacles, e.g., state-transfer requirements, result availability limitations, and performance implications. Being separating from the agreement stage, some replicas may advance at different speeds and this phenomenon can cause up to f execution replicas to be far behind on the process of execution. Also, the execution stage typically requires that at least $f + 1$ replicas in order to reach a consistent checkpoint. Thus, the state-transfer protocol between agreement and execution stages must provide information which enables replicas to verify the correctness of a checkpoint. This requires some special mechanisms to guarantee the stability of checkpoints. For example, a proof of a checkpoint should not only contain the information of the checkpoint itself, but also enable correct replicas to resort to the checkpoint if the proof is valid, so that all other replicas can get the consistent state. In general, this can be done by assembling a checkpoint certificate and constructing a hash of the checkpointed state with the help of a digital signature [176]. Also, the resulting availability must be guaranteed, so that all correct clients have the ability to access these checkpoints and restore their internal state to a consistent state. To guarantee the resulting availability, one possible and simple solution is to extend checkpoints to attach the latest reply to each client, as illustrated in solutions [216] [305]. In these solutions, the checkpoint plays a key role in proceeding to consensus, in which it must contain all information to help a correct replica go back to a correct state. This would definitely increase the complexity of the checkpoint. Besides, separating execution replicas eliminates the shared information with the agreement stage, and the replicas in the execution stage would require extra efforts to prepare additional information. This would increase some overhead, and thus lower the performance.

Another advantage of separating the execution stage is to enable the relaxed execution order. In a classic BFT replicated system (e.g., performing agreement and execution stages on the same replica), replicas need to sequentially process requests, which may significantly limit performance. To address this issue, many efforts have been developed to relax the order in which the replicas execute requests. Roughly speaking, these efforts can be classified into two categories: non-speculative execution and speculative execution. In general, non-speculative execution can guarantee a consistent state when applied by some correct replica, while speculative execution may result in inconsistencies which further require a correct replica to rollback its state.

In general, non-speculative executions focus on the order of state-object accesses. One way to achieve this is that all accesses to a state object is protected by some *locking* scheme. For example, OptSCORE [306] relies on a customized scheduler to carry out a deterministic order-execution on requests, with the help of sequentializing the execution process only on some critical sections of the operations and enabling the paralleling execution on the rest of the operations. However, this scheme needs some kind of modifications on the application code to fit the non-speculative execution, e.g., introducing the lock/unlock primitives during the access to state objects. Another example of non-speculative execution is the CBASE [307], in which a parallelizer component is introduced between the agreement stage and the execution stage on each replica. It only works in the case that both the agreement stage and execution stage are performed on the same replica. The parallelizer is used to perform analysis on the sequence of requests agreed from the agreement stage and to decide which requests can be executed safely and parallelly without involving conflicts on these parallel executions. In general, the parallelizer must be equipped with some application-specific knowledge, e.g., structure, characteristics, and possible dependencies of requests [6]. On the other hand, the speculative execution can simplify the reasoning procedure on the correctness of these executions and be targeted to improve execution efficiency and performance. In general, this approach may introduce somewhat temporary inconsistencies in replicas' internal execution states even among correct replicas. However, these inconsistent states will be finally resolved before replying to the client, and the system never releases these internal states to clients or other external processes. Also, this kind of execution would rely on some rollback scheme to ensure replica has the ability to reset its state to a consistent state. Thus, the request execution should not rely on any irreversible actions, e.g., remote function calls to a process outside of the system or sending any control signals/internal states to critical infrastructure (as this may involve some wrong actions before obtaining a correct final result). As long as these internal states are erasable and kept within the replicated system, an application can adopt this speculative execution to speed up the process and improve responsiveness. The well-known example in this approach is the Zyzzyva [59].

D. Checkpointing

With the increase of the requests submitted by clients, the size of the confirmed sequence of requests on each replica will exceed the amount of memory to proceed with further requests at some point. Thus, replicas must resort to some form of garbage collection scheme to remove the state information on already processed entries [6]. This will assist replicas to make progress on request processing. However, the lack of additional mechanisms on this cleaning would pose a problem for asynchronous BFT systems. For instance, some correct replicas may fall a lot behind other replicas, even with a full-functioning garbage collection scheme; also, some correct replicas may not be restored back to a correct state as the synchronization messages may fail to get the right point to restore. This will create an indefinite stuck in the progress. To resolve this issue, correct replicas can periodically generate checkpoints of all its internal states and only garbage-collect data that are already in a stable state (e.g., with a threshold number of replicas agreements or endorsements). This also enables some slow replicas (e.g., fallen behind others) to catch up and update their states according to some well-established checkpoints. A checkpointing scheme not only helps replicas recover from faults, but also can allow new replicas to join a running system with little effort. In general, the checkpointing scheme may involve what kinds of contents should be checkpointed, how to represent checkpoints, as well as the corresponding creation process.

In general, a checkpoint must contain all information that reflects the replica state at a specific time (e.g., the proof on the execution of a specific request). In general, the information contained in a checkpoint should be indistinguishable in order for correct replicas to recover to a state. The checkpointed information typicaly consists of three parts: application state, protocol state, and replies. The application state in a checkpoint provides information on the objects representing the state of the replica's application instance, which helps replicas to execute the application-specific requests during the execution stage. This kind of state information is typically application-specific with its internal structure being independent of its consensus protocol. The protocol state in a checkpoint consists of the application-independent state and the state required by the BFT replication protocol, in order to offer certain consistency guarantees. This kind of state information is protocol-specific. The replies in a checkpoint should contain the information replied on processed client requests, which allows a correct replica to reply to the client request in a case that its previous tries failed (e.g., due to unreliable links). All the above three states are useful to construct a checkpoint on a replica's state. However, according to different BFT replicated systems, the checkpoint of a replica does not necessarily contain all these three parts. Instead, a practical checkpoint protocol often aims to create a "good enough" state without endangering the guarantees provided by the overall system.

The representation of checkpoints typically can be presented in two different forms: state checkpoints [15] [172] [308] [305] and hybrid checkpoints [198]. State checkpoints highly rely on the sequence number when the checkpoints are created to reflect the corresponding version. Hybrid checkpoints allow a replica to generate some temporary checkpoints frequently with reference to infrequent replica snapshots (e.g., stable state), in which a replica only needs to keep the latest infrequent snapshot and apply the requests that have been executed between these two checkpoints. In general, the creation of checkpoints would definitely involve substantial overhead. Based on these checkpoints representation, there are several checkpoint creation approaches to trade off the complexity and efficiency, namely, stop and copy, copy on write, checkpoint rotations, and deterministic fuzzy checkpoints [6]. The first two approaches are straightforward approaches from their names. The stop and copy approach suspends the execution of the request, creates the checkpoint, and then resume the request execution [198], while the copy-on-write approach does not require a stop to minimize the downtime overhead and only process to the actually changed parts since the last checkpoint [15] [172]. The checkpoint rotation approach simply rotates the responsibility for generating the checkpoint among replicas to mitigate the overall server disruption. It does not require a replica to produce a snapshot for every checkpoint interval and for every sequence number; instead, each replica only performs the snapshot for some sequence numbers. However, this approach may introduce some vulnerability, e.g., a manipulated state transfer by malicious replicas [308]. The fuzzy checkpoint approach is more flexible, which allows correct replicas to independently choose the time point to start capturing the data for the next checkpoint with a guarantee to generate identical checkpoints [308]. A replica in this approach can start earlier to proceed with the state object even before obtaining an agreed sequence number on requests. It offers slow replicas the chance to start early in an effort to complete the checkpoint so that they can catch up to a similar point as the fast replicas.

E. Recovery

Byzantine replicas can behave arbitrarily. For BFT systems, without proper measurements, the effects of faults can be accumulated over time (especially under a strong adversary), which may eventually lead to a violation of the predefined tolerance threshold and break the system. And, a replica recovery mechanism is required to recover replicas from faults. In general, a good recovery scheme allows a replicated system to successfully recover solely based on its own available information, so that it can guarantee that the number of faulty replicas does not exceed the pre-designed threshold. From the literature, there are two major approaches to perform recovery, namely, *reactive recovery* and *proactive recovery*. Reactive recovery happens when some faults are detected, while proactive recovery happens at some pre-established points in time (e.g., periodical time). Also, a practical replicated system can combine both approaches [309].

In general, a reactive recovery requires a BFT system to have mechanisms for detecting faulty replica behaviors, e.g., by analyzing the outputs of a replica and comparing them with the outputs of other replicas. In this case, it may involve the participation of clients to show some evidence to replicas that faults happen. However, involving clients in a faulty detection process may introduce some level of inaccuracies, as clients typically do not know what happened in a replicated system, e.g., clients cannot distinguish a faulty replica or a correct slow replica in case not enough replies are received. In general, the reactive recovery approach can be applied for two scenarios by observing the system status [190] [309]: that the faulty behavior has been provably detected (e.g., providing incorrect result) or that it only has been suspected (e.g., timeout happening). A proactive recovery scheme can effectively mask faults without affecting the interaction with clients, as this approach is performed on a regular basis, instead of relying on some evidence of fault defection scheme [15]. Proactive recovery is performed on a regular basis which may incur some unnecessary overhead on recovering correct replicas. A recovery approach typically is application-specific, and there is no universal recovery scheme [310]. Both approaches have their advantages and shortcomings. Interested readers can refer to the corresponding papers.

Practically, a recovery scheme should have the ability to transmit from a faulty state to a fault-free state. A typical recovery process consists of three steps: system reboot, replica rejoin the group, and state update [15]. After a faulty replica is detected, restarting the replica's machine can erase existing faulty processes and some potential effects of instruction. In general, after rebooting, a replica's logic can be considered correct, while its state still may be corrupted, missing, or out of data, which requires some configuration. When rejoining a replica group, a replica is required to re-establish the connections with both clients and other replicas in the system. The state update step is used to update its state with other replicas once again being able to communicate. The goal of this step is to enable a replica to once again actively participate in the replication protocol as a correct replica. Besides these steps for a recovery scheme, according to practical application scenarios, consideration of other aspects may be needed, e.g., replica group reconfiguration [266].

This section discusses some basic components to build a BFT protocol in the form of modulization. The system designers can combine different approaches in each module to construct a customized BFT protocol. There are many similar literature works on modulization design, and detailed information can be found in the corresponding works, e.g., [16] [17] [299] [6].

VI. Blockchain Consensus Algorithms

Blockchain as a distributed and decentralized database is a chain of blocks in which each block contains a list of well-ordered transactions. The process to reach an agreement among participants of a blockchain network is complex and thus important. Consensus plays a key role in most distributed systems, including blockchain. Most existing protocols are designed to solve consensus problems under various assumptions. With recent advances in blockchains, various consensus protocols have been proposed and studied to reach an agreement among participating nodes so that they can be applied to different application scenarios. Byzantine fault tolerance protocol is one of such protocols which can be used to resolve consensus issues in the blockchain. BFT consensus protocols can be considered as the classic consensus to establish an agreement even under Byzantine replicas. Instead of discussing these classic Byzantine-based consensus protocols (as shown in Section IV), this section discusses some important consensus algorithms which specifically target blockchain networks. We can collectively call them "*PoX*" (Proof-of-X), where "*X*" is some measurable criteria and can be used as a proof in blockchain networks.

A. PoX

Many existing PoX protocols are proposed for various applications. We will briefly discuss these general design principles.

1) Proof-of-Work (PoW): The most well-known consensus protocol for blockchain is the proof-of-work (PoW) protocol, which is widely adopted in Bitcoin [5] and many other crypto-currencies. Due to the popularity of Bitcoin, we also commonly call the PoW protocol Nakamoto consensus, however, PoW itself has been proposed much earlier than Bitcoin. The key idea of PoW was first proposed by Dwork and Naor in 1992 as a mechanism to handle spam mails [311], in which the email senders must first work on a mathematical puzzle to show that they performed some computational work before sending an email. And later, another system for fighting spam [312] was independently proposed for Hashcash by Back in 1997. In the Hashcash scheme, the email sender was required to compute a computational puzzle by applying an SHA-hash on the content email (including its recipient's address and email itself) with some special requirements, e.g., the hash must contain at least a certain number of leading *zeros*. Due to the property of pre-image resistance of hashing algorithms, the task can only be done by repeated tries to find out some qualified random *nonces* to meet the leading zero requirements. The key idea of Nakamoto consensus literally comes from Hashcash, by replacing the SHA-1 hash with two successive SHA-2 hashes, and demands that a qualified hash should below some target integer value t. And this makes the difficulty of the puzzle adjustable, e.g., decreasing the target value t would increase the amount of "work" to compute a qualified hash. Thus, by changing these pre-defined conditions, the network can be very scalable and flexible to any condition. In PoW of Bitcoin, the computing nodes typically are called *miners* and the process to compute a valid hash is referred to as *mining*. In Bitcoin, miners calculate hashes of candidate blocks of transactions as a kind of proof, when one node finds the target value, it would broadcast the block to the whole network and all other nodes must confirm the correctness of the hash value. Once the block is validated and gets confirmed, the mining node gets rewarded with new coins, and all nodes append this new block to their own chain [18].

Although the PoW algorithm provides high security and decentralization, the function of mining and validating blocks wastes a lot of energy, and its speed and success rate highly depend on the computational abilities of the hardware performing the hash operations. Solving the puzzle typically takes some time, which is not suitable for large and fast-growing networks that require high throughput. Besides, it is also subject to various attacks, e.g., chain forks, double-spending attacks, 51% attacks, and mining centralization.

2) Proof-of-Stake (PoS): Compared to PoW, PoS protocols replace wasteful computations with useful "work" derived from alternative commonly accessible resources. The key idea of a PoS algorithm is that the creator of a block is chosen by various combination features, such as the amount of stakes and the ages of these stakes. In general, PoS algorithms can provide some level of scalability. This method does not require high computational resources for validating any proof, which strongly depends on nodes that have the most stake and the blockchain will somehow become centralized. More specifically, the PoS algorithm randomly elects "leaders" from the qualified stakeholders (such as the ones with minimal stakes in the qualified group), then the elected leaders can have a chance to append their generated blocks to the blockchain. In general, PoS has a candidate pool which contains all qualified stakeholders (e.g., the amount of stakes is larger than a threshold value) [1]. The leader election process is a critical part of PoS, and the leader election process can be either public or private. In a public leader election, the result of the election process is visible to all the participating nodes [313] [314], so that in a private leader election process, its final result can be verified by all other participants simply by resorting to publicly available information [315]. In general, the private leader election process can resist DoS attacks since it requires candidates to privately check if they were elected to propose the next block before actually sending out their blocks, and this makes it too late to launch DoS attacks. There are several recent PoS-based systems which have proved secure, e.g., Ouroboros [314], Ouroboros Praos [315], and Snow-White [313]. Ouroboros belongs to the class of publicly leader elections, in which a random cluster of qualified stakeholders together engage into a multiparty coin-tossing protocol to generate a random seed. This random seed is then used as a pseudo-random function to elect the leader among the qualified stakeholders. To guarantee fairness, Ouroboros distributes rewards (e.g., from transaction fees) among all participants no matter whether they were elected as a leader or not. The participants in Ouroboros Praos independently determine if they have been elected, so their system belongs to the class of the private leader election. This is with the help of a verifiable random function (VRF) to generate a random value. If the generated random value is below some predefined value, it implies that the corresponding participant is selected as a leader who has a chance to broadcast its block (with proof of qualification) to the network. Snow-White uses a Bitcoin-like mechanism to elect a leader, in which the participants are required to compute a pre-image hash below some pre-defined target. Each participant is only allowed to compute one hash per time step with the help of a random oracle. Also, it considers the amount of stake of the participants. One of the challenges in PoS is that it is hard to trace and keep up to date the actual stake changes on each stakeholder.

Although PoS provides some benefits compared with PoW, it is subject to some new types of attacks, e.g., *nothing-at-stake attack*, *grinding attack*, and *long-range attack* [316]. The nothing-at-stake attack means a node has nothing at its stake while misbehaving, and the node is not afraid of losing anything. As stated in [18], it does not require too much effort to extend a chain, when forking happens, some rational miners may engage to mine on each qualified forking chain so that they increase the chance to get their blocks into the final chain. One straightforward way to mitigate this kind of attack is to introduce some penalty schemes to penalize the attempts to fork. The grinding attack means a miner re-creates a block multiple times until it is likely that the miner can create a second block shortly afterward. In general, this kind of attack can be limited by utilizing an unbiasable and unpredictable random number generator to generate randomness so that a miner cannot influence the selection of the next leader. The long-range attack means that an attack may actively bribe miners to exchange their old private keys, and typically there is no cost or little cost for bribed miners to sell their old keys to an attacker. Even though these keys are old keys, they indeed had some value in the past, and the attacker may use these keys to re-mine previous blocks, which has some potential to re-write the whole block history of the blockchain. In general, this kind of attack can be limited by using some central checkpointing scheme, for instance, some entities can actively declare old blocks are in the final state if sufficient time has passed.

We should mention that PoS is not just one but instead a collection of protocols. There exist many PoS alternatives, which require *miners* to hold or prove the ownership of assets.

Delegated-Proof-of-Stake (DPoS): DPoS is based on PoS, in which the nodes select representatives through voting to validate blocks [317]. The number of representatives is limited and this makes it possible to organize the network more effectively and each representative can determine the adequate time to publish each block. In general, it provides some important features, such as scalability, energy efficiency, and low-cost transactions. However, this method introduces a level of centralization, thus DPoS is more suitable for private blockchains.

Proof-of-Burn (PoB): In PoB, miners have to burn some of their already owned cryptocurrencies to get rewards, in which burning means that a user is required to send some cryptocurrencies to an "eater address" to receive rewards, e.g., by sending them to a verifiably unspendable address [318]. The cryptocurrency sent to that address is unrecoverable and no one can spend it again, so it is called *burnt* and is out of circulation. Slimcode [318] implemented this approach in 2014, which burns Bitcoin as a mining method. The more coins a user burns, the more chances to find the next block.

Proof-of-Deposit (PoD): In PoD, miners are required to 'lock' a certain amount of assets, and these locked assets cannot be spent during their mining process. In general, the locked assets in PoD are a kind of proof that coins have been deposited into an account, which typically is used for finance sectors. For example, Tendermint [238] utilizes a similar mechanism, in which miners' voting power proportionately relates to the number of assets being locked.

Proof-of-Importance (PoI): In PoI, each node is assigned an importance score which influences the node's chance of getting a small financial reward in exchange for adding users' transactions to the network. The NEM project [319] utilizes this scheme to solve the incentive issues in PoS.

3) Proof-of-Elapsed-Time (PoET): PoET is based on trusted hardware (e.g., the enclave in Intel SGX) to provide proof of these "efforts" [320]. PoET consensus was developed in 2016 by Intel. It provides an advanced tool for solving the computational problem, namely "random leader election". In general, it requires the participants to wait some time in their enclaves, and the one with the shortest time wins and elects as a leader. The proof typically is provided by the enclave which has a tamper-resistant proof, called attestation. This attestation, along with a new block, can be used to attest two things: (1) it indeed has the shortest wait time, and (2) it indeed waits the designated waiting time before broadcasting its block. Thus, the PoET algorithm is a lottery-like system. Although, like PoW, in PoET the participants must be equipped with some types of hardware processors, it does not require processors to perform cryptographic hash operations, an uses less power. Also, PoET allows the miner's processor to take a snooze or sleep and switch to another task for a particular period, which increases its efficiency.

4) Proof-of-Capacity: In PoC, miners use the free spaces on their hard disk to mine free coins. The success of voting new blocks is mainly based on their own capacity and the actual amount of disk space that they assign to the mining process. There are two main application scenarios currently based on the PoC scheme: PermaCoin [321] and SpaceMint [322]. In PermaCoin, participants must store pieces of a large file, and the more a participant stores, the more chance it has to add a new block. In general, there exists one authoritative entity to sign and distribute this large file. Typically, the file can be recovered by the participating node in case that the owner of the file is shut down or fails. SpaceMint utilizes a consensus scheme that is based on a non-interactive variant of PoC (namely proof-of-space) to show the available space. One of the key problems in PoC protocols is that it is subject to centralization. This is because the participants are required to outsource some file storage provided by an external authority.

B. Discussion on PoX

In PoX-based blockchain, forks may happen when two individual participants successfully issue distinct blocks (on different transaction lists) based on the same previous block. They need extra schemes or rules to resolve this equivocation and get an agreement among all participants. For example, in Bitcoin, forks (e.g., double-spending attack) are settled by accepting the "longest chain" rule, in which a longer chain represents the greater PoW effort that has been invested into that chain. Based on the longest chain rule, with the time increase, a final chain will be selected from forks, and thus the case of double-spending can be settled. Also, most PoX schemes require that more than 51% of 'X' is occupied by honest participants, otherwise, the security of the whole network would be compromised. Another common problem in PoX is that it is easy to form a mining pool to aggregate resources and share rewards. However, this subverts the feature of decentralization and introduces some level of censorship if the pool manager behaves maliciously. This also requires some extra techniques to mitigate it. For example, SmartPool [323] tries to achieve a decentralized mining pool manage scheme with the help of Ethereum smart contracts, in which the smart contract actively takes the role of pool manager in a traditional scheme.

Technically speaking, PoX is not a decent consensus protocol, whose mechanism is often used for determining the membership of participants in a Sybil-attack-resistant fashion. Due to some historical reasons, e.g., Bitcoins used PoW as a "consensus" protocol to build a Bitcoin blockchain, we literally categorize them into consensus protocols. For example, in a hybrid consensus (e.g., ByzCoin [241] and Hybrid Consensus [248]), the decent consensus protocol (the algorithm for agreement on shared history) is separable from and orthogonal to the membership Sybil-resistance scheme (e.g., PoW). Another example is the sharding scheme for blockchain [1]. Typically, a sharding scheme uses a Byzantine replicated protocol as its consensus, with the help of PoX. Roughly speaking, both protocols have different tasks in an overall sharding scheme. PoX is typically used for committee formation to establish the committee members and their corresponding identities, while BFT is used for the intra-committee consensus, which is used within a committee to form the blocks.

C. PoW vs. BFT Blockchain

Both PoX and BFT can be used as the components of a consensus to achieve replication consistency in distributed systems. BFT algorithms as a generic scheme to handle consistency provides two main principles [324]: a) *Non-equivocation* (e.g., preventing equivocating behavior of leaders so that each time only one proposal comes out) and 2) *Proposal-safety* (e.g., replicas should achieve a consistent state among all honest replicas). PoX and BFT achieve these two replication principles in a different way, which results in BFT having instant finality and PoX not having it. In general, BFT-based blockchain offers good performance for a small number of replicas, while when that number extends to large scale, e.g., hundreds of replicas, BFT protocols are often subject to scalability issues. In contrast, PoX-based blockchains offer good nodes scalability, but with poor performance. In this part, we use PoW as a representative of PoX, and make a comparison with BFT consensus. Following the basic comparison of PoW and BFT consensus [19], we extend some new features. Table II provides a high-level comparison for both PoW and BFT when applying them as blockchain protocols. These properties are not completely exhaustive, but it is enough to represent both BFT-based and PoW-based blockchain families. In general, given the inherent features of both BFT and PoW, it is not clear which techniques are optimal for blockchain for many use cases when the number of participating nodes ranges from a few tens to the level of a few thousands [19].

Both PoX and BFT have their advantages and disadvantages, and we can combine PoX with BFT as a hybrid solution. One target of such hybrid solutions is to combine their advantages and overcome their deficiencies. For example, PoX-based blockchain replication often suffers from a long-term deficiency in which there exists a risk that a block will not be committed. This can typically raise many concerns and risks, i.e., if an adversary may later obtain enough resources (e.g., computational power) and redo all previous transactions. BFT protocols can help to resolve these issues. Actually, one of the most important advantages of BFT protocol to the blockchain is that it can provide "*instant finality*", a property that a PoX protocol does not achieve. That means, once a block is successfully committed, that block is finalized and cannot be revoked. However, one big issue is that the BFT replicated system works well only for a small number of replicas. When scaling to a large scale (e.g., thousands of replicas), it may come with some critical challenges [324].

Abraham and Malkhi [324] present several approaches to combine both PoX and BFT techniques to overcome each other's challenges. The first approach works like Ethereum Casper [325] by setting up a group of trusted validators. The validators then validate PoX blocks later (e.g., after a certain amount of time or a certain amount of block depth) by performing a BFT protocol, and only validated blocks can be viewed as formally committed. The second approach is to select a rotating committee by using PoX, and the selected committee makes the final decision to append the block to the chain. Several literature utilize this approach, such as Bitcoin-NG [244], ByzCoin [241], and Hybrid Consensus [248]. A third approach, as shown in Solidus [326], constructs a blockchain directly by using the customized consensus and without involving any longest chain rule. That means, the new block is generated solely via the underlying consensus to append blocks to the chain. Also, there exist some randomized sample schemes to select a sub-committee (e.g., the work on scalable leader election [327]). However, such schemes require hiding the committee selection process against a targeted attack (e.g., on the committee members). Algorand [121] utilizes both PoS and VRF to secure the committee selection process, which can mitigate the targeted vulnerability of committee members.

TABLE II. HIGH-LEVEL COMPARISION BETWEEN POW AND BFT PROTOCOLS AS BLOCKCHAIN CONSENSUS [19].

	Pow Consensus	***BFT Consensus***
Node Identity Management	Open, fully decentralized	Permissioned, knowing neighbors' IDs
Consensus Finality	Longest chain rule	Yes
Scalability (# of Nodes)	Excellent (Thousands of nodes)	Limited (Variarty from tens to hundreds)
Scalability (# of Clients)	Excellent (Thousands of clients)	Excellent (Thousands of clients)
Performance (Throughput)	Limited (Chain forks)	Excellent (Tens of thousands tx/sec)
Performance (Latency)	High latency (Multi-block confirmations)	Excellent (Match network latency)
Tolerated Power of an Adversary	$\leq$25% computing power	$\leq$ 33% voting power
Network Synchrony Assumptions	Physical clock timestamps (for block validity)	None for consensus safety (Synchrony needed for liveness)
Correctness Proofs	No	Yes
Parallel Execution	No	Yes
Colluding Possibility	Easily form centralized pool	Hard
Common Attacks	Chain fork (Double-spending, Sybil attacks)	Byzantine nodes (Abitrary behavior)
Privacy-preserving	Anonymous identity	Hard to achieve
Common Application Scenarios	Crypto-curriencies	Industrial use cases

VII. CHALLENGES ON APPLYING BFT TO BLOCKCHAIN

Blockchain as a distributed and decentralized ledger technology provides several key properties, such as decentralization, immutability, transparency, and trustworthiness. However, this technology is still in its infancy stage, in which blockchain technology faces multiple challenges and problems to overcome.

A. Blockchain Challenges

Typically, the robustness and strength of blockchain highly depend on the consensus algorithm that is used to get an agreement among participants. This section lists some critical blockchain challenges in general, which in turn may affect the design of its consensus protocol.

1) Vulnerability: In general, cybersecurity attacks theoretically can threaten almost all types of consensus algorithms. And there are also other special types of attacks and vulnerabilities for blockchain protocols, e.g., double-spending attack, 51% attack, and Sybil attack. These kinds of attacks can specifically be designed for both PoX and BFT protocols.

a) Double-spending attack: A double-spending attack occurs when a user tries to spend the same assets or resources twice or more, which should be spent only once in practice. This could happen when an attacker attempts to create a transaction and include that transaction into a block, however, after some time, the attacker constructs a conflicting transaction (using the same assets of its first transaction) and pushes this conflicting transaction into a new forked block. This attempt will try to revert to the transaction previously made. That means, the attacker already obtains the result of the first transaction before the majority declare that the second transaction is invalid. The attacker tries to extend the forked branch of blockchain until the forked chain accepts the conflicting transaction as a correct one [328] [329]. Various research efforts have been attempted to mitigate the double-spending attack, however, this attack cannot be completely prevented and it can occur at any time [330].

b) 51% attack: In a blockchain network, a minimum of 51% of the nodes is required to approve and validate any transaction. This type of attack was first exploited by Bitcoin's PoW blockchain network. Theoretically, the 51% attack is not avoidable [331], which means it may not be able to be completely prevented. When an attacker has the ability to control and manage more than 50% of resources in the blockchain system, it has the ability to perform some malicious activities, such as double-spending or preventing other nodes' progress. In general, the attacker does not require that it always have more than 50% of the resources of the network, and the attacker can temporarily use these resources (e.g., via renting or bribing other nodes) to launch an attack. Practically, when we say 51% attack, it may specifically target a PoX consensus protocol. BFT consensus protocols are also subject to this kind of attack, e.g., in the case that more than $1/2$ of replicas are Byzantine.

c) Sybil attack: In a Sybil attack, the attacker tries to create a number of fraudulent or fake identities, which are used to affect the correct functioning of a network. These identities appear to be unique nodes that are in fact in control by the attacker, and are used to gain some voting power (if behaving benignly) or broadcast some fake message (if behaving maliciously). For example, in a generic BFT protocol, we typically assume all replicas are the same and only put some constraints on the ratio of faulty replicas in the total number of replicas. Thus, it is easy to launch a Sybil attack. A successful Sybil attack can grant the attacker disproportionate control over the network or surround an honest node and try to influence the information reaching it, which further influences the blockchain. Sybil attacks are typically hard to identify and prevent. For blockchain, there are some counter approaches to prevent them, e.g., 1) increasing the cost of creating a node, 2) requiring some type of trust, and 3) giving unequal power to the identities. Each approach has its own advantages and disadvantages, and there is no "one-fit-all" solution. For example, for increasing the cost of creating a node, it is a challenge to find the ideal cost for the identity creation without restricting normal nodes from joining the network. The approach of requiring some type of trust may involve a centralized scheme, which obeys the decentralized design principle of blockchain. And the approach of giving unequal power to the identities may make the system a meritocracy instead of a pure democracy and may not be interesting for new users [332].

2) Scalability: The key features of the blockchain (e.g., decentralization and immutability) require that every full node store a full copy of the blockchain; however, this comes at a cost of scalability. The scalability issue in blockchain limits its wide usage in large-scale networks. Typically, scalability can be evaluated by the *throughput* (e.g., measured by transactions per second) against the number of nodes and the number of concurrent workloads [333] [334]. In the current design, many blockchain systems are still suffering from poor throughput. Scaling blockchain has become an active research area, for example, via increased block size [335] and sharding techniques [1]. Blockchain scalability issues are still an open research area, and many different initiatives and efforts in recent research are aimed at improving blockchain scalability, from side chains to sharding techniques.

In general, these efforts to address the scalability problems can be roughly categorized into two categories: storage optimization of blockchain and redesigning blockchain structure [336]. In storage optimization, old transactions are deleted from the network, as it is not necessary for network nodes to store all transactions to validate a transaction and the recent ones have more value [337]. The work on Chainsplitter [338] belongs to this category. Also, a lightweight client can also benefit from this scheme, without involving large communication to transmit these old transactions. For redesigning blockchain, the blockchain can adopt different structures, such as side chains and sharding, to parallel the process of transactions. In general, parallelism can handle the scalability issues in blockchain, but, its security may be weakened. Thus, it is often necessary to evaluate the trade-off between scalability and security.

3) Privacy Leakage: In general, it is hard to guarantee transactional privacy in public blockchain as all transactions must be publicly accessible and visible. Similar to security, blockchain technologies have some mechanisms for preserving a certain degree of privacy for transactions recorded in blockchain (e.g., via anonymous identity technology). However, these adopted privacy-preserving technologies in current blockchain systems are not robust enough. For example, attackers can track the user's IP address [339], and a privacy breach can occur by drawing interference based on a graph analysis of network nodes with which a user transacts [340] [341]. A better solution for preserving privacy in blockchain would be in a form of decentralized record-keeping that is completely obfuscated and anonymous by design. Several techniques can be used to mitigate privacy issues in blockchain, such as a ring signature [342] and address mixing [343] [344]; however, when applying these techniques directly to different domains, they are also subject to other critical issues (e.g., resource constraint issues in performing complex computations in IIoT devices).

B. Integrating BFT to Blockchain

State-machine replication with BFT bring many advantages for both distributed and decentralized systems. BFT protocols can naturally be integrated into blockchain as its consensus protocol, taking the advantage of instant finality. Compared with PoX-based consensus, once a block under BFT consensus is committed, it can never be revoked, which is different from probabilistic finality (e.g., the inverse resistance relying on the deep on the chain). And modern BFT SMR variants already have achieved very low latency and high throughput, especially in the common and failure-free cases. Although BFT protocols have been more mature and efficient, there still exist some challenges when applying them to blockchain scenarios [18] [345] [324]. For example, the permissionless model of PoX allows them to naturally fit a much larger set of replicas, while scaling BFT-based solutions (e.g., to hundreds or even thousands of replicas) raises some new challenges. Instead of providing detailed and quantified analysis on BFT protocols, this section provides some discussion on integrating BFT into the blockchain, from high-level perspectives.

1) Permission Management: BFT protocols in nature are subject to scalability issues, and many recent scalable BFT-based blockchain protocols have been proposed under different assumptions, such as RSCoin, Omniledger, RapidChain, and Chainspace. Essentially, these protocols employ traditional Byzantine consensus protocols for scalability with the help of sharding technologies [1]. Although these protocols achieve high throughput and low latency, they are inherently in 'closed' settings, in which replicas typically have stable and authenticated channels for communication, and only authorized members can participate in the construction of the blockchain. The consensus protocol in their prototypes cannot accommodate open participation of replicas and are more vulnerable to Sybil attacks. Also, some scalable protocols have synchrony assumptions on message transmission to achieve the features of safety and liveness. Some newer BFT protocols, e.g., HoneyBadgerBFT, try to include a randomized consensus algorithm to achieve these features in an asynchronous scenario. They still do not resolve the issue of the need for a 'closed' group with strict permission management. And this prevents them from being used in a permissionless blockchain (like the case of PoW consensus). Also, the communication and computational overheads in these randomized BFT protocols are much higher than that of deterministic ones. It is still an open research problem for Byzantine consensus protocols under the open group participationship.

2) Identity Management: For some BFT protocols (e.g., without a digital signature), it is possible that malicious members create some spoofed messages to fake these messages generated from correct members and try to bias the consensus process. In general, BFT protocols apply permission and identity management to prevent this from happening. For example, a typical BFT replication system assumes point-to-point and authenticated channels between all replicas, with some mechanisms to track committee members and their keys (i.e., public key). However, tracking membership and key distribution in dynamic permissionless committees is a challenging task, which makes it difficult to adopt in an open blockchain. Although there exist some naive solutions, e.g., all replicas periodically broadcast their identities to the entire network, this typically results in $O(n^2)$ communication complexity. Some approaches [230] provide a special committee to offer directory services to new committee members, however, this static committee will undermine decentralization, and a decentralized discovery committee will suffer heavy communication complexity. Thus, adopting the BFT consensus protocol into the blockchain requires resolving the issues in identity management.

3) Primitive Management: Most existing mature blockchain projects are still based on traditional BFT protocols, e.g., BFTSmart library, with message complexity $O(n^2)$ (where n is the number of participating nodes). Practically, using some hardware or cryptographic primitives can effectively reduce this message complexity. For example, ByzCoin leverages a scalable collective signing scheme and communication trees to reduce its common case consensus to $O(n)$. Some consensus protocols are leveraging trusted platform modules, e.g., the enclave of Intel SGX, to create secure hardware execution environments. This essentially can provide a secure execution environment on ordering transactions, which can further reduce the number of communicated messages, without compromising safety and liveness. Although these primitives can effectively improve communication efficiency, they put some extra burdens and costs on replicas. For example, some replicas are lightweight, and not all replicas can afford these novel and expensive primitives.

VIII. Discussion and Future Directions

This section presents some discussion on applying BFT protocols to blockchains, and provides some future directions.

A. Choices on Paxos vs. BFT

Paxos is a well-known consensus protocol, which can achieve an agreement under crash failures [52]. Originally, it was proposed to circumvent the FLP impossibility. In general, Paxos can forgo progress under temporary asynchrony, while when the system goes back to synchrony, it continues to work and keep the system consistent. By forgoing the process, Paxos keeps the system consistent during asynchronous periods. Also, it can tolerate c crash faults out of $2c + 1$ replicas. There are three types of replicas in a classic Paxos protocol, namely, *proposers*, *acceptors*, and *learners*. And they run in a sequence of ballots, which is different from the rounds in BFT protocols. Compared with BFT, Paxos is much simpler. The replicas do not have to go through rounds, and piggybacking more than one round and role at once, efficiently eliminates most of the communication overhead. The key idea behind Paxos is that the replicas do not perform operations arbitrarily, which means they are either honest or crashed. Also, there exist several derivatives of Paxos, including multi-Paxos [346] and fast-Paxos [136]. Even though these derivatives can achieve the same fault tolerance rate (i.e., c out of $2c+1$), they in general are more complex. For example, multi-Paxos protocols require a repeated application to run Paxos, and this would increase the complexity of the design. Besides, there are many Paxos-based projects for blockchain applications, e.g., Raft and Tendermint. Raft consensus is based on Paxos, whose basic idea is that nodes collectively select a leader and the rest of the nodes become followers. The leader is used to maintain the consistent state, e.g., via state transition logs, across its followers.

Crash faults typically are easy to detect and tolerate, while Byzantine faults may behave arbitrarily which is indistinguishable from correct replicas. For blockchain applications, Byzantine replicas have more power to bias the consensus protocol, and maintaining consensus is vital to a blockchain network [347]. From this perspective, it is better to apply BFT consensus to the blockchain. However, for some specific replicated systems, such as datacenters, it may be good to apply Paxos-based consensus protocols, due to their simplicity.

In general, classic Paxos (or more general CFT) and BFT explicitly model machine faults only, and they can be combined with an orthogonal network fault model (i.e., synchronous model and asynchronous model). Thus, the scope can be roughly classified into four categories [10]: synchronous CFT [11] [25], asynchronous CFT [25] [52] [12], synchronous BFT [20] [13] [30], and asynchronous BFT [15] [14]. According to different blockchain applications, the system designers can choose its right consensus protocols. Besides, there exists some hybrid fault models, e.g., XFT [10], Byzantine Paxos [210], and heterogeneous Paxos [348], to handle both CFT and BFT.

B. Hybrid Fault Models

Byzantine fault model makes consensus protocol inherently difficult to develop. Typically, a BFT system may assume a powerful adversary or harsh network conditions, or even combine both, which comes with a cost on complexity and overhead to design a well-replicated system [6]. However, some designers have observed that it is not worth designing such Byzantine replicated systems for some secure and reliable systems, e.g., the use cases of datacenters [310] [234]. And some recent works have moved to hybrid fault models [349] with weaker guarantees, e.g., Byzantine replicas only accounting for a very small portion in all faulty replicas, to achieve a practical implementation. There are several literature works on these hybrid fault models, e.g., UpRight [198], VFT [234], and XFT [10]. Upright separates crash faults from incorrect behaviors, and each faulty type has an upper boundary. Its replicated system has fewer replicas as not all faulty replicas would perform incorrect behavior. VFT further weakens this hybrid model aiming to minimize the number of replicas in the system, with the assumption that there exists a well-organized communication graph among replicas. However, if this assumption does not hold, the system may develop unsafe or unavailable conditions. Similarly, XFT also relies on a hybrid fault model, whose faults include crashed faults, incorrect behaviors, and partitioned replicas. In general, these protocols can work well under some relatively secure and predictable environments.

Under hybrid fault models, trust plays an important role to get replicated systems to work well. In general, a trusted system equips a small trusted computing base [350] which makes it possible to identify incorrectness. A malicious replica may have the ability to operate on untrusted components, but does not have the ability to control trusted components. With the advances of modern processors, it is better to implement the trust parts in dedicated hardware modules, such as trusted platform module (TPM) [351] [81], Intel's SGX [352], and ARM's TrustZone [353], to provide trusted execution environments. Also, there are some solutions to establish trusted parts in software, e.g., via the proxy [354], a multicast ordering service [355] [16], or a virtualization layer [304] [356] [303]. In general, trusted components can ensure replicas are recovered even if they are compromised [15]; also they can be used to prevent a faulty leader from successfully performing equivocation. Thus, trusted components, either in the form of hardware or software, offer some level of trustworthiness under the hybrid fault models, which can help replicated systems to reach a consensus with fewer required replicas. Also, this hybrid fault module is more practical in some application scenarios, such as datacenters and permissioned blockchain systems.

C. Liveness in Consensus

A BFT consensus protocol typically makes progress in a sequence of *views*, each with a designed leader to drive the whole consensus process. Liveness is one of the two key properties (w.r.t. safety and liveness) that consensus wants to achieve, which is used to ensure that a transaction sent to all honest validators will eventually be executed. Historical reasons have made researchers pay more attention to safety and less attention to liveness, e.g., PBFT provides an explicit safety proof and its liveness is treated via a non-trivial scheme. However, achieving the liveness of BFT consensus is no less challenging than its safety [357]. The result of FLP impossibility makes it impossible to ensure both safety and liveness when there exist faulty replicas under asynchronous network settings. And most existing BFT consensus schemes target guaranteeing safety in all scenarios (e.g., both synchronous and asynchronous settings), and its liveness can only be ensured during some synchronous periods. One key observation to achieve liveness is to make all correct replicas stay on a view with a correct leader, so that the leader can have enough time to make a decision. In this process, the view leader plays a key role to make the decision. However, if a decision is not eventually reached, e.g., due to timeout or a malicious leader, the correct followers together switch to the next view leader and continue the consensus process.

Theoretically, these consensus protocols can achieve liveness, with the assumption of some unknown *Global Stabilization Time (GST)*. For example, after some GST period, the network may go to a period of synchrony, e.g., with a bounded but unknown constant message delay. However, most existing work claiming a liveness guarantee fails to provide a concrete value (e.g., latency) on this bound to make a decision. Thus, in practice, as the communication is asynchronous, the participants cannot return at the exact same time in all processes. This makes the system subject to attacks. For example, if some correct processes are still at the beginning of their round while an adversary observes the result of other participants for the same round, then the adversary can prevent progress among the correct processes by controlling messages between correct processes and by sending specific values to them. Even if a correct process invokes some correct results before the Byzantine process, the Byzantine still can prevent a correct process from progressing [358].

In the literature, there exist some works which suggest relaxing the liveness issues under various network conditions. For example, Abraham et al. [270] use the concept of clock synchronization [359] [360] to achieve a “view synchronization", in which each correct replica can access hardware clocks with reliable and bounded drift on time. The HotStuff protocol [284] utilizes a component called *PaceMaker* to achieve view synchronization and advance progress, but it fails to provide a detailed specification on how to achieve this functionality. Bravo et al. [357] presents a similar view synchronization scheme to provide a wrapper for the functionality of BFT consensus procedures, and it provides formal specification only under partial synchrony. Although they made much progress on releasing the liveness of BFT protocols, from a high-level perspective, there is still no practical live Byzantine consensus working under fully asynchronous environments like the Internet. Thus, there is a long way to go getting a safe and live BFT consensus protocol.

D. Privacy in Consensus

When mentioning the term “privacy", we typically refer to its meanings at the application level instead of in an abstract system level. For instance, we can enhance privacy in blockchains, while it is hard to enhance privacy on consensus protocols. As we discussed in Section V, BFT consensus protocols have stages of agreement and execution. In the agreement stages, they can utilize some level of obfuscation without revealing the information of requests, e.g., using agreement on sequence numbers. However, during the execution stage, accessing the requested information for verification purposes cannot be avoided. This presents huge challenges to privacy-preserving different aspects, e.g., clients, clients' requests, and their meta-data. Practically, it is hard to achieve privacy-preserving in BFT consensus protocols. However, privacy can be achieved at the application level, e.g., via the property of “anonymity". And, we should note that some low level pseudonymity for privacy-preserving can be easily circumvented by tracking attacks [361].

In general, a permissioned system provides some level of protection of privacy by restricting its participants. For instance, BigchainDB [362] restricts the set of core participants in the consensus to a small vested set, which are assumed to be trusted on both integrity and liveness of systems, and also can be trusted for keeping secrets (privacy) [18]. However, this scheme itself cannot prevent information leakage. For example, since the information is replicated, any participants may violate the property (i.e., semi-honest or malicious participants), and there is no practical scheme to detect this kind of violation as it just leaks secrets without performing malicious behaviors. Also, relying on a permissioned ledger for privacy forces the whole system to rely on closed groups, which makes the design choice hard to change. There are indeed some protocol-level techniques to provide privacy-preserving, e.g., non-interactive zero-knowledge proof (zk-SNARK) [363] or smart contracts [364]. However, protocol-level techniques for privacy-preserving are only used for the specific context of applications.

Ideally, privacy-preserving systems wish to disclose little information to others for verification purposes. For example, for blockchain, each replica may want to agree only on the order of transactions for execution, and the creation and execution of these transactions take place off-chain, with each party having full access to its own information [18]. In general, to achieve privacy-preserving in a complex blockchain system with desired properties, there are some remarks to follow, which also suit blockchain security [365]. 1) No single technique fits all for the privacy of blockchain, and the appropriate privacy techniques should be chosen based on the privacy requirements and the context of applications. In general, the combination of multiple technologies exhibits more efficiency than solely relying on a single technology. 2) No technique has no defects or is perfect in all aspects, and new techniques may cause or involve some new forms of attacks. 3) Privacy is not cost free, and there typically exist some level of trade-offs between privacy and system efficiency. The system designers must achieve a balance between privacy and other system properties.

E. Gossip Protocols

Gossip protocols are typically used to distribute and share information between different parties within a network, and their main purpose is for information dissemination and aggregation among many nodes. Compared with broadcast or multicast protocols, one major advantage of gossip protocol is its logarithmic mixing time, in which it can propagate a well-organized information to all peer nodes only in $O(logn)$ time, where n is the number of peers [366]. Also, gossip protocols are very simple and straightforward, where all peers have the same piece of code to run. All these features are ideally suitable for BFT protocols to lower the communication complexity. A simple gossip protocol typically consists of three processes in series, namely, *peers selection process*, *data exchange process*, and *data processing process* [367]. In general, each node in a gossip protocol simply forwards all its received messages to a set of randomly selected peers. Under this simple design, it can be adapted to large-scale networks with guarantees of fast and reliable information dissemination [368]. In general, gossip communication protocols can be used to manage inconsistencies that arise in distributed systems, which are simple to implement and highly resistant to failure. A replicated system can converge to a consistent state using a gossip protocol, despite temporary partitions and process failures [369].

Most existing BFT consensus protocols in blockchains assume a peer-to-peer communication channel, and their analysis on communication complexity is based on this assumption. However, in a practical distributed network, pure peer-to-peer communication is rare. Besides, when applying BFT to a blockchain system, a major problem is that membership is closed. Gossip protocols can be easily adopted to open and dynamic networks, and membership information also can be gossiped along with other messages [370]. An ideal gossip protocol would send fewer messages per second and the messages sent over the network can be quite large, depending on the number of nodes. However, due to its intrinsic randomness on peer selection and variety on message transmission latency, it is hard to estimate an upper boundary for the duration of the protocol. A practical design often requires membership to be closed for the duration of the protocol, which limits its application scenarios [371]. In the literature there exist several recent blockchain works utilizing gossip protocols for message dissemination (with the underlying BFT protocols), e.g., Algorand, OmniLedger, and RapidChain. However, these gossip-based protocols are highly redundant and random, as each node may receive the same message from distinct peers multiple times. This would definitely involve more transmitted messages and degrade the network performance. Thus, efficient gossip protocols are required in dissemination networks.

Many existing blockchain protocols rely on some customized gossip protocols to share and exchange information (e.g., transactions, blocks, and memberships) among participating parties [372]. In general, gossip protocols can ensure that each party receives all messages with a high probability (instead of in a deterministic way). There exist several critical questions on utilizing gossip protocol to reduce communication complexity. Gossip protocols are inherently susceptible to data corruption, which means the gossip process is subject to Byzantine attack. For example, it is necessary to address the issue of when and how participating nodes choose to end up agreeing on a block. Also, involving incentive mechanisms for message passing in the construction of blockchain makes it impossible to prove which participants (and in what level of efforts) have actively contributed to the gossip protocol. Even some trusted hardware platforms (such as Intel SGX) can help, but this will in turn increase the deployment cost. Currently, a gossip-based blockchain is just a preliminary protocol, and many design spaces for blockchains need to be explored. Also, when adopting gossip protocols to permissionless scenarios, extra efforts (e.g., security analysis and refinement) will be required to ensure that gossip-based blockchain solutions are safe and trusted for special-purpose applications (e.g., healthcare systems with privacy-preserving and industrial infrastructure with cyber-security) [373].

F. Scalability

Most BFT protocols are limited in their scalability, either in terms of network size (e.g., number of nodes) or the overall throughput. The design space for improving them is vast. We will use Practical BFT (PBFT) [4] as an example to explain BFT scalability. The original PBFT protocol requires at least $n = 3f + 1$ nodes to tolerate up to f Byzantine faults. It has been shown not to scale beyond a dozen nodes due to its quadratic communication complexity [333]. Typically, scaling protocols for BFT focus on either reducing the number of nodes required to tolerate f Byzantine faults [144], [154], or reducing the protocol's communication complexity to allow larger network sizes [241].

a) Reducing the number of nodes: To tolerate f Byzantine nodes that can *equivocate* in a *quorum* system like PBFT, quorums must be intersected by at least $f + 1$ nodes [72]. Consequently, if a BFT protocol requires $n = 3f + 1$, its quorum size is at least $2f + 1$. The smaller n means the lower communication cost incurred in tolerating the same number of faults; it also means that for the same number of nodes n, the network can tolerate more faulty nodes. One way to reduce the number of nodes is to randomly select a small set of consensus nodes, as a committee, to run a consensus process. A smaller consensus committee can lead to better throughput, as a smaller committee attains higher throughput due to lower communication overhead. Sharding technology reduces the consensus process within one shard. However, in this scenario, the security of each shard, e.g., the ratio of the number of faulty nodes to the size of a shard, will be a top concern. It can be mitigated by utilizing some mechanisms, e.g., the epoch randomness, to guarantee a "good majority" for each shard with a high probability [230].

Another way to reduce the number of nodes is to utilize techniques to get down the n from $3f + 1$ to $2f + 1$. Those techniques are mainly based on leveraging external components (e.g., the trusted hardware) or lessening the system models. For example, *BFT-TO* [374], a hardware-assisted Byzantine replicated protocol, demonstrates that it is possible to use only $2f + 1$ replicas to tolerate f Byzantines with the help of trusted distributed components. Similarly, there exist a few other algorithms to achieve the consensus with less replicas, such as A2M-BFT-EA [144], MinBFT [81], MinZyzzyva [81], EBAWA [351], CheapBFT [161], and FastBFT [155]. Besides, there also exist some other work to achieve the same purpose by lessening the system models. For example, the work in [375] improves the BFT threshold to $2f + 1$ by utilizing a relaxed synchrony assumption.

b) Reducing communication complexity: PBFT protocol has been perceived to be a communication-heavy protocol. There is a long-standing myth that BFT is not scalable to the number of participants n, since most existing solutions incur the message transmission of $O(n^2)$, even under favorable network conditions. As a result, existing BFT chains involve very few nodes (e.g., 21 in [376]). Even with reduced network size, PBFT still has a communication complexity of $O(n^2)$. Byzcoin [241] proposed an optimization wherein the leader uses a collective signing protocol (CoSi) [242] to aggregate other node's messages into a single authenticated message. By doing so, each node only needs to forward its messages to the leader and verify the aggregate message from the latter. In this way, by avoiding broadcasting, the communication complexity is reduced to $O(n)$. Besides, there is some work [377] on utilizing trusted execution environments (TEEs) (e.g., Intel SGX [378]) to scale distributed consensus. TEEs provide a protected memory and isolated execution so that the regular operating systems or applications can neither control nor observe the data being stored or processed inside them [379]. Generally, trusted hardware can only crash but not be Byzantine. However, introducing trusted hardware into consensus nodes is expensive, and specific knowledge is needed to implement the protocol. Similarly, the security in this category can be mitigated by using cryptographic primitives, such as threshold signatures [274] [380].

G. Performance Optimization

With the recent growth of blockchain technology and its applications, a variety of complex BFT consensus algorithms have been developed with diverse properties. Surprisingly, even after a decade of development, the main use cases for blockchain still lie in financial sectors, especially for crypto-currencies. Currently, there has been a pretty slow adoption for both blockchains and BFT in practical applications [381]. One obvious reason is its slow throughput compared with prior works [382] [383] [384] achieving a throughput of the order 100K transactions per second. The low throughput of the blockchain (or BFT) limits its development and wide adoption. Thus, it is critical to find out some important criteria that affect the performance of these algorithms. Instead of discussing the properties from a high-level perspective, e.g., scalability, safety, and liveness, we here identify some important performance criteria from a user perspective, e.g., throughput and responsiveness. These performance criteria highly affect the user experience, and thus the market cap of usage. To be consistent with blockchain, we interchangeably use the terms *transactions* in blockchain to represent the *requests* in the consensus protocol.

In general, the throughput of a consensus algorithm is measured by the maximum rate of agreement on values done. It means, the maximum throughput is the maximum rate at which the blockchain can confirm requests, which can be roughly measured by transaction per second (TPS). However, if using batch processing, it also relates to the maximum batch size. TPS is defined as the number of transactions executed per second, which is the number of transactions that occur in one second through an information system. And, TPS measurement can be further used to calculate the performance of systems that handle routine transactions and record-keeping jobs, as well as to determine the speed of the platform in the execution of transactions. Typically, the higher the number of TPS, the faster the transactions will be executed, validated, and confirmed on the platform. Besides TPS, throughout also depends on other parameters, such as agreement latency, verification latency, and patch size. According to the modulated consensus protocol, consisting of agreement and execution phases (See Section V), agreement latency and execution latency include the time spent in the agreement and execution stages, respectively.

We consider a consensus protocol as responsive protocol if it commits and responds to clients' requests at the level of the actual network latency, without being thwarted by pre-defined system parameters [247]. From a user perspective, responsiveness can be literally understood as the time taken to confirm their transactions without caring about the underlying processes. For example, original Bitcoin may take 10 mins to get confirmed, and takes more than one hour to get the final confirmation. Also, without considering responsiveness, it may cause serious worries of performance degradation. As seen in the findings of Dolev-Strong [13], almost all deterministic protocols definitely involve some delay on confirmation time even under the synchronous network. In general, asynchronous consensus protocols can take some advantages of responsiveness without any form of timing assumptions [112]. This is because asynchronous protocols are responsive by design, whereas synchronous protocols are not [258]. Some modern asynchronous consensus protocols can achieve their responsiveness in seconds under normal cases (e.g., without faulty nodes or faulty leader [284]); however, when there exist faulty nodes or when applying to large-scale geo-scale distributed systems, their responsiveness greatly degrades. Practically, geo-scale consensus typically distinguishes local and global communication and, using topological information, all replicas in a single region can be grouped into a single cluster [385]. By doing so, some level of optimistic responsiveness can be achieved. Thus, there is still a long way to go to improve consensus performance.

Besides, there exist other promising techniques to optimize performance, e.g., via parallelization. The main idea of parallelization is to allow non-conflicting transactions to execute in parallel, and resort to a fork resolution scheme to get a final consistent block. Many BFT researches have been extensively exploring the idea of parallel replication, leveraging parallelization of execution of independent requests [300] [386].

H. Practical Implementation

Due to the low performance of BFT consensus processes in the aspects of throughput and responsiveness, currently, it does not catch public perception and BFT-based blockchain applications show a slow adoption. Even with several decades of developments, few of them have been applied to practical projects, and the popularly deployed BFT consensus protocols in blockchain are still PBFT and BFTSmart. For example, Hyperledger Fabric utilizes PBFT variant (Apache Kafka) [86] as its consensus, and Hyperledger Indy utilizes Redundant BFT (RBFT) [165] as its consensus. The industry-grade permissioned blockchain systems only select a group of users (some of which may not be trusted) to participate, e.g., Hyperledger Fabric [387]. However, the throughput of current permissioned blockchain applications is still of the order of 10K transactions per second [387] [388] [389], and the case for permissionless is much less than permissioned cases. Several works [333] [59] [389] [284] blame the lower throughput and scalability of blockchain on its underlying BFT consensus protocols.

As pointed out by Gupta [381], there exist several factors to affect BFT large-scale adoption, namely, single-threaded monolithic design, successive phases of consensus, decoupling ordering, and execution, strict ordering, off-memory chain management, and expensive cryptographic practices. All those factors can highly affect the performance of a BFT-based blockchain system, further affecting its wide adoption. Currently, besides blockchain applications in finance (i.e., crypto-currencies), there still lack applications in other application areas, especially in industrial use cases, e.g., Internet of Things (IoT) and industrial IoT, public and social services. IoT typically connects physical things to the Internet and provides services to the end-users. Its killer applications include RFID technology, smart homes, e-health, etc. Industrial IoT typically targets industrial sectors, including smart manufacturing, supply chain, smart grid, food industry, etc. The public and social services are more targeted to life-related matters, including land registration, energy-saving, education, etc. All those application domains can benefit from blockchain, and it is promising to apply blockchain into these domains. In general, when considering applying blockchain to these applications, BFT protocols provide more benefits than PoX protocols, such as instant finality. Thus, the first priority is to design and implement some customized and efficient BFT protocols. There is still a long way to go to achieve that.

I. Other Topics

Blockchain has shown great potentials in both industry and academia, and BFT-based consensus protocols play a very important role in successfully building a blockchain ecosystem. To successfully adopt BFT replicated systems into the blockchain, it also needs to incorporate other technologies, e.g., smart contract, big data analytics, AI, to facilitate more promising features. Most of these technologies can be considered as a wrapper to pre-process the raw data before submitting requests to BFT replication systems. In general, these kinds of technologies do not affect the progress of the BFT consensus process, however, they indeed can provide some optimization. For example, big data analytics and AI can help the load-balancing on requests when multiple consensus groups exist at the same time.

In general, a smart contract is a digitized transaction protocol that can execute the terms of contracts automatically once the specified clauses are met [390]. When applied to the blockchain era, a smart contract is a code fragment that can be executed by participating parties automatically, which can significantly reduce the involvement and intervention of human beings. With more and more smart contracts emerging, it helps to extend blockchain from current financial cases into many new areas, e.g., commercial IoT and industrial IoT [36]. Meanwhile, a smart contract can be modeled as a state machine, whose consistent execution across multiple nodes in a distributed environment can be achieved using SMR [19]. However, one critical thing needed to carefully take care of is safety and security, as smart contracts (as a piece of code) are subject to various attacks, e.g., DAO attack in 2016 [391].

Blockchain can also benefit from big data, e.g., data management and data analytics. For the use cases in data management, blockchain can be used as a medium to record important data with guarantees on the immutability of these data. However, less important data can be stored off-chain and provide an index to the main blockchain. Also, blockchain can be used for data analysis by exploring the transactions on the blockchain. For instance, by analyzing blockchain transactions, some common patterns might be extracted to predict users' behavior. Using these patterns, a blockchain system may eventually work more efficiently. Further, game theory is a powerful tool for strategic decision-making, which also can be useful for blockchain. It can be used to optimize the utility of each participating party while considering the interactions among parties [392]. However, the game theory model often assumes the deviating coalitions are rational to optimize their utility, which is different from a malicious model which can deviate parties arbitrarily [324].

AI technologies also could help blockchain to solve many challenges. When applying smart contracts into the blockchain, there always must be an oracle to respond for determining whether the specified conditions are met. And, in general, this oracle is controlled by a third party [336]. For instance, some chaincodes provided in HyperLedger Fabric can be only operated by a trusted entity. We can apply AI technology to create an intelligent oracle, which is in a decentralized version and no single party has the power to control it. This oracle can learn from historical data and train itself, which makes the smart contract smarter. AI techniques, such as machine learning algorithms, are powerful for analyzing and optimizing blockchain operations. Combining these two technologies can be a game-changer for the next generation decentralized Internet.

Besides, there are some other interesting research topics, e.g., testing technology to evaluate the efficiency of both BFT protocols and blockchain, schemes to prevent malicious replicas collaboration or centralization. There is still a long way to go for both BFT consensus protocols and blockchains.

IX. Conclusion

Within recent years, the researches on BFT consensus have a dramatic surge partially due to the emergence of blockchain. This paper presents a Systematization of Knowledge for the existing efforts on BFT consensus protocols. We carefully studied the selected BFT protocols and tried our best to provide a comprehensive review with detailed analysis. This paper can serve as a starting point for exploring consensus in the areas of both BFT and blockchain. Based on what we observed and learned, we discussed opportunities and challenges when applying BFT protocols into the current blockchain design. Finally, we provide several potential research directions that can help to advance reliable and robust BFT consensus for the blockchain ecosystem.

References

[1] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 41–61.

[2] D. Malkhi and M. Reiter, "Byzantine quorum systems," in *STOC*, vol. 97. Citeseer, 1997, pp. 569–578.

[3] C. Cachin and M. Vukolić, "Blockchains consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[4] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2008.

[6] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–38, 2021.

[7] O. Maric, C. Sprenger, and D. Basin, "Consensus refined," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 391–402.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one fault process." YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, Tech. Rep., 1982.

[9] J. Aspnes, "Randomized protocols for asynchronous consensus," *Distributed Computing*, vol. 16, no. 2-3, pp. 165–175, 2003.

[10] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić, "{XFT}: Practical fault tolerance beyond crashes," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 485–500.

[11] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," *Information and Computation*, vol. 118, no. 1, pp. 158–179, 1995.

[12] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988, pp. 8–17.

[13] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[14] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 363–376.

[15] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[16] M. Correia, G. S. Veronese, N. F. Neves, and P. Verissimo, "Byzantine consensus in asynchronous message-passing systems: a survey," *International Journal of Critical Computer-Based Systems*, vol. 2, no. 2, pp. 141–161, 2011.

[17] C. Berger and H. P. Reiser, "Scaling byzantine consensus: A broad analysis," in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2018, pp. 13–18.

[18] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 183–198.

[19] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International workshop on open problems in network security*. Springer, 2015, pp. 112–125.

[20] L. LAMPORT, R. SHOSTAK, and M. PEASE, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[21] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[22] L. Lamport, "The weak byzantine generals problem," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 668–676, 1983.

[23] J. Sousa and A. Bessani, "From byzantine consensus to bft state machine replication: A latency-optimal transformation," in *2012 Ninth European Dependable Computing Conference*. IEEE, 2012, pp. 37–48.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications*, 1978.

[25] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[26] Q. Zhang, Z. Qi, X. Liu, T. Sun, and K. Lei, "Research and application of bft algorithms based on the hybrid fault model," in *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*. IEEE, 2018, pp. 114–120.

[27] V. Gramoli, "From blockchain consensus back to byzantine consensus," *Future Generation Computer Systems*, vol. 107, pp. 760–769, 2020.

[28] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[29] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[30] P. Berman, J. A. Garay, K. J. Perry *et al.*, "Towards optimal distributed consensus," in *FOCS*, vol. 89. Citeseer, 1989, pp. 410–415.

[31] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, "Distributed consensus protocols and algorithms," *Blockchain for Distributed Systems Security*, vol. 25, 2019.

[32] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 77–97, 1987.

[33] A. Sunyaev, "Distributed ledger technology," in *Internet Computing*. Springer, 2020, pp. 265–299.

[34] M. S. Ali, M. Vecchio, M. Pincheira, K. Dolui, F. Antonelli, and M. H. Rehmani, "Applications of blockchains in the internet of things: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1676–1717, 2018.

[35] R. Lai and D. L. K. Chuen, "Blockchain–from public to private," in *Handbook of Blockchain, Digital Finance, and Inclusion, Volume 2*. Elsevier, 2018, pp. 145–177.

[36] G. Wang, "Sok: Applying blockchain technology in industrial internet of things."

[37] ——, "Sok: Exploring blockchains interoperability." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 776, 2021.

[38] M. S. Ferdous, M. J. M. Chowdhury, M. A. Hoque, and A. Colman, "Blockchain consensuses algorithms: A survey," *arXiv preprint arXiv:2001.07091*, 2020.

[39] J. Garay and A. Kiayias, "Sok: A consensus taxonomy in the blockchain era," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 284–318.

[40] W. Simpson, *RFC1661: the point-to-point protocol (PPP)*. RFC Editor, 1994.

[41] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[42] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 931–948.

[43] M. Okun, "Agreement among unacquainted byzantine generals," in *International Symposium on Distributed Computing*. Springer, 2005, pp. 499–500.

[44] E. A. Alchieri, A. N. Bessani, J. da Silva Fraga, and F. Greve, "Byzantine consensus with unknown participants," in *International Conference On Principles Of Distributed Systems*. Springer, 2008, pp. 22–40.

[45] A. Beimel and M. Franklin, "Reliable communication over partially authenticated networks," *Theoretical computer science*, vol. 220, no. 1, pp. 185–210, 1999.

[46] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[47] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[48] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[49] R. Pass and E. Shi, "The sleepy model of consensus," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 380–409.

[50] C. Dwork and Y. Moses, "Knowledge and common knowledge in a byzantine environment: crash failures," *Information and Computation*, vol. 88, no. 2, pp. 156–186, 1990.

[51] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[52] L. LAMPORT, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[53] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[54] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[55] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[56] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Byzantine fault detectors for solving consensus," *The Computer Journal*, vol. 46, no. 1, pp. 16–35, 2003.

[57] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for byzantine fault detection." in *HotDep*, 2006.

[58] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 59–74, 2005.

[59] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[60] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[61] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 177–190.

[62] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, pp. 1–39, 2010.

[63] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire." in *NSDI*, vol. 8, 2008, pp. 189–204.

[64] B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012.

[65] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: a scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 568–580.

[66] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[67] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[68] M. K. Reiter, "The rampart toolkit for building high-integrity services," in *Theory and practice in distributed systems*. Springer, 1995, pp. 99–110.

[69] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The securering group communication system," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 4, pp. 371–406, 2001.

[70] C. Cachin and J. A. Poritz, "Secure intrusion-tolerant replication on the internet," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 167–176.

[71] K. Kursawe, "Optimistic byzantine agreement," in *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* IEEE, 2002, pp. 262–267.

[72] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.

[73] D. Malkhi and M. K. Reiter, "An architecture for survivable coordination in large distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 187–202, 2000.

[74] G. Chockler, D. Malkhi, and M. K. Reiter, "Backoff protocols for distributed mutual exclusion and ordering," in *Proceedings 21st International Conference on Distributed Computing Systems*. IEEE, 2001, pp. 11–20.

[75] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal byzantine storage," in *International Symposium on Distributed Computing*. Springer, 2002, pp. 311–325.

[76] L. Zhou, F. B. Schneider, and R. Van Renesse, "Coca: A secure distributed online certification authority," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 329–368, 2002.

[77] C. P. Fry and M. K. Reiter, "Nested objects in a byzantine quorum-replicated system," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* IEEE, 2004, pp. 79–89.

[78] F. Muratov, A. Lebedev, N. Iushkevich, B. Nasrulin, and M. Takemiya, "Yac: Bft consensus algorithm for blockchain," *arXiv preprint arXiv:1809.00554*, 2018.

[79] L. J. Gunn, J. Liu, B. Vavala, and N. Asokan, "Making speculative bft resilient with trusted monotonic counters," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 133–13 309.

[80] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin, "Revisiting fast practical byzantine fault tolerance," *arXiv preprint arXiv:1712.01367*, 2017.

[81] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2011.

[82] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 4, pp. 1–45, 2015.

[83] C. Y. da Silva Costa and E. A. P. Alchieri, "Diversity on state machine replication," in *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2018, pp. 429–436.

[84] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 169–180.

[85] G. Mack Diouf, H. Elbiaze, and W. Jaafar, "On byzantine fault tolerance in multi-master kubernertes clusters," *arXiv e-prints*, pp. arXiv–1904, 2019.

[86] J. Sousa, A. Bessani, and M. Vukolic, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2018, pp. 51–58.

[87] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From byzantine replication to blockchain: Consensus is only the beginning," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 424–436.

[88] T. Swanson, "Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems," *Report, available online*, 2015.

[89] E. Androulaki, C. Cachin, K. Christidis, C. Murthy, B. Nguyen, and M. Vukolic, "Next consensus architecture proposal. hyperledger wiki, fabric design documents," 2016.

[90] Symbiont, "Symbiont assembly platform," *URL https://www.symbiont.io/technology/assembly*, 2016.

[91] D. Mohanty, "Corda architecture," in *R3 Corda for Architects and Developers*. Springer, 2019, pp. 49–60.

[92] N. Rakotondravony and H. P. Reiser, "Visualizing bft smr distributed systems-example of bft-smart," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 152–157.

[93] J. Niu and C. Feng, "Leaderless byzantine fault tolerant consensus," *arXiv preprint arXiv:2012.01636*, 2020.

[94] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain," *The Libra Assn., Tech. Rep*, 2019.

[95] L. Lamport, "Brief announcement: Leaderless byzantine paxos," in *International Symposium on Distributed Computing*. Springer, 2011, pp. 141–142.

[96] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman, "Cogsworth: Byzantine view synchronization," *arXiv preprint arXiv:1909.05204*, 2019.

[97] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," *arXiv preprint arXiv:1906.08936*, 2019.

[98] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "Dbft: Efficient leaderless byzantine consensus and its application to blockchains," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–8.

[99] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous binary byzantine consensus with t< n/3, o (n2) messages, and o (1) expected time," *Journal of the ACM (JACM)*, vol. 62, no. 4, pp. 1–21, 2015.

[100] B. Arun, S. Peluso, and B. Ravindran, "ezbft: Decentralizing byzantine fault-tolerant state machine replication," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 565–577.

[101] N. Shrestha and M. Kumar, "Revisiting ezbft: A decentralized byzantine fault tolerant protocol with speculation," *arXiv preprint arXiv:1909.03990*, 2019.

[102] A. Gągol, D. Leśniak, D. Straszak, and M. Świętek, "Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 214–228.

[103] A. Gągol and M. Świętek, "Aleph: A leaderless, asynchronous, byzantine fault tolerant consensus protocol," *arXiv preprint arXiv:1810.05256*, 2018.

[104] L. Bonniot, C. Neumann, and F. Taïani, "Pnyxdb: a lightweight leaderless democratic byzantine fault tolerant replicated datastore," in *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2020, pp. 155–164.

[105] P. Raykov, N. Schiper, and F. Pedone, "Byzantine fault-tolerance with commutative commands," in *International Conference On Principles Of Distributed Systems*. Springer, 2011, pp. 329–342.

[106] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[107] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 265–278.

[108] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, and M. I. Zablotchi, "Leaderless consensus," EPFL, Tech. Rep., 2021.

[109] E. Gafni, "Round-by-round fault detectors, unifying synchrony and asynchrony (extendeda abstract)," in *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing (PODC), Puerto Vallarta, Mexico, June*, 1998, pp. 143–152.

[110] M. O. Rabin, "Randomized byzantine generals," in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 1983, pp. 403–409.

[111] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 27–30.

[112] Y. Lu, Z. Lu, and Q. Tang, "Bolt-dumbo transformer: Asynchronous consensus as fast as pipelined bft," *arXiv preprint arXiv:2103.09425*, 2021.

[113] R. Gelashvili, L. Kokoris-Kogias, A. Spiegelman, and Z. Xiang, "Be prepared when network goes bad: An asynchronous view-change protocol," *arXiv preprint arXiv:2103.03181*, 2021.

[114] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 2005, pp. 191–201.

[115] M. Ben-Or, B. Kelmer, and T. Rabin, "Asynchronous secure computations with optimal resilience," in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994, pp. 183–192.

[116] F. Shen, Y. Long, Z. Liu, Z. Liu, H. Liu, D. Gu, and N. Liu, "A practical dynamic enhanced bft protocol," in *International Conference on Network and System Security*. Springer, 2019, pp. 288–304.

[117] J. Baek and Y. Zheng, "Simple and efficient threshold cryptosystem from the gap diffie-hellman group," in *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, vol. 3. IEEE, 2003, pp. 1491–1495.

[118] C. Delerablée and D. Pointcheval, "Dynamic threshold public-key encryption," in *Annual International Cryptology Conference*. Springer, 2008, pp. 317–334.

[119] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[120] A. Mostefaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous byzantine consensus with t< n/3 and o (n2) messages," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014, pp. 2–9.

[121] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 51–68.

[122] J. Chen and S. Micali, "Algorand," *arXiv preprint arXiv:1607.01341*, 2016.

[123] ——, "Algorand: A secure and efficient distributed ledger," *Theoretical Computer Science*, vol. 777, pp. 155–183, 2019.

[124] A. Spiegelman and A. Rinberg, "Ace: Abstract consensus encapsulation for liveness boosting of state machine replication," *arXiv preprint arXiv:1911.10486*, 2019.

[125] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[126] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[127] P. Li, G. Wang, X. Chen, F. Long, and W. Xu, "Gosig: a scalable and high-performance byzantine consensus for consortium blockchains," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 223–237.

[128] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[129] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 129–138.

[130] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[131] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Verifying distributed erasure-coded data," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 139–146.

[132] M. Castro and B. Liskov, "Proactive recovery in a byzantine-fault-tolerant system," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, 2000.

[133] ——, "Byzantine fault tolerance can be fast," in *2001 International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 513–518.

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

[135] ——, "Fast byzantine paxos," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004, pp. 402–411.

[136] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing paxos," *ACM Sigact News*, vol. 34, no. 1, pp. 47–67, 2003.

[137] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[138] R. Van Renesse, C. Ho, and N. Schiper, "Byzantine chain replication," in *International Conference On Principles Of Distributed Systems*. Springer, 2012, pp. 345–359.

[139] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, vol. 4, no. 91–104, 2004.

[140] H. Mendes and M. Herlihy, "Multidimensional approximate agreement in byzantine asynchronous systems," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 391–400.

[141] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 33, no. 3, pp. 499–516, 1986.

[142] I. Abraham and D. Malkhi, "Bvp: Byzantine vertical paxos," *Proceedings of the Distributed Cryptocurrencies and Consensus Ledger (DCCL), Chicago, IL, USA*, vol. 25, 2016.

[143] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009, pp. 312–313.

[144] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 189–204, 2007.

[145] I. Abraham, M. K. Aguilera, and D. Malkhi, "Fast asynchronous consensus with optimal resilience," in *International Symposium on Distributed Computing*. Springer, 2010, pp. 4–19.

[146] M. Garg, S. Peluso, B. Arun, and B. Ravindran, "Generalized consensus for practical fault tolerance," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 55–67.

[147] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[148] V. Shoup and R. Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1998, pp. 1–16.

[149] C. Stathakopoulou, T. David, and M. Vukolić, "Mir-bft: High-throughput bft for blockchains," *arXiv preprint arXiv:1906.05552*, 2019.

[150] T. Crain, C. Natoli, and V. Gramoli, "Evaluating the red belly blockchain," *arXiv preprint arXiv:1812.11747*, 2018.

[151] Z. Milosevic, M. Biely, and A. Schiper, "Bounded delay in byzantine-tolerant state machine replication," in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 2013, pp. 61–70.

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

[153] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, vol. 9, 2009, pp. 153–168.

[154] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: Sgx-based high performance bft," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 222–237.

[155] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139–151, 2018.

[156] J. Li, M. N. Krohn, D. Mazieres, and D. E. Shasha, "Secure untrusted data repository (sundr)." in *Osdi*, vol. 4, 2004, pp. 9–9.

[157] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small trusted hardware for large distributed systems." in *NSDI*, vol. 9, 2009, pp. 1–14.

[158] S. L. Kinney, *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[159] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6. Berkeley, CA, USA, 2003, pp. 68–72.

[160] M. Ryan, "Introduction to the tpm 1.2," *DRAFT of March*, vol. 24, 2009.

[161] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: Resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 295–308.

[162] M. Eischer and T. Distler, "Scalable byzantine fault tolerance on heterogeneous servers," in *2017 13th European Dependable Computing Conference (EDCC)*. IEEE, 2017, pp. 34–41.

[163] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.

[164] J. Zhang, J. Gao, K. Wang, Z. Wu, Y. Lan, Z. Guan, and Z. Chen, "Tbft: Understandable and efficient byzantine fault tolerance using trusted execution environment," *arXiv preprint arXiv:2102.01970*, 2021.

[165] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 297–306.

[166] R. Rodrigues, M. Castro, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 15–28, 2001.

[167] M. Castro, R. Rodrigues, and B. Liskov, "Using abstraction to improve fault tolerance," in *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 27–32.

[168] B. Liskov and J. Guttag, *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education, 2000.

[169] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*. IEEE, 1995, pp. 381–390.

[170] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, vol. 1, 1978, pp. 3–9.

[171] J. Gray and D. P. Siewiorek, "High-availability computer systems," *Computer*, vol. 24, no. 9, pp. 39–48, 1991.

[172] M. Castro, R. Rodrigues, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 236–269, 2003.

[173] R. Baldoni, J.-M. Helary, and M. Raynal, "From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 2000, pp. 273–282.

[174] A. Doudou, B. Garbinato, and R. Guerraoui, "Encapsulating failure detection: From crash to byzantine failures," in *International Conference on Reliable Software Technologies*. Springer, 2002, pp. 24–50.

[175] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[176] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 253–267.

[177] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "Bar fault tolerance for cooperative services," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 45–58.

[178] N. Ntarmos and P. Triantafillou, "Aesop: Altruism-endowed self-organizing peers," in *International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2004, pp. 151–165.

[179] J. Shneidman and D. C. Parkes, "Specification faithfulness in networks with rational nodes," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, 2004, pp. 88–97.

[180] A. Clement, H. Li, J. Napper, J.-P. Martin, L. Alvisi, and M. Dahlin, "Bar primer," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 287–296.

[181] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "Bar gossip," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 191–204.

[182] D. C. Oppen and Y. K. Dalal, "The clearinghouse: A decentralized agent for locating named objects in a distributed environment," *ACM Transactions on Information Systems (TOIS)*, vol. 1, no. 3, pp. 230–253, 1983.

[183] J. Li and D. Mazieres, "Beyond one-third faulty replicas in byzantine fault tolerant systems." in *NSDI*, 2007.

[184] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating Systems Principles*, 2007, pp. 59–72.

[185] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Low-overhead byzantine fault-tolerant storage," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 73–86, 2007.

[186] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient byzantine-tolerant erasure-coded storage," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 135–144.

[187] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 197–206.

[188] ——, "Prime: Byzantine replication under attack," *IEEE transactions on dependable and secure computing*, vol. 8, no. 4, pp. 564–577, 2010.

[189] C. Ho, R. Van Renesse, M. Bickford, and D. Dolev, "Nysiad: Practical protocol transformation to tolerate byzantine failures." in *NSDI*, vol. 8, 2008, pp. 175–188.

[190] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: Practical accountability for distributed systems," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 175–188, 2007.

[191] A. R. Yumerefendi and J. S. Chase, "The role of accountability in dependable distributed systems," in *Proceedings of HotDep*, vol. 5. Citeseer, 2005, pp. 3–3.

[192] I. T. Downard, "Simulating sensor networks in ns-2," NAVAL RESEARCH LAB WASHINGTON DC, Tech. Rep., 2004.

[193] N. M. Preguiça, R. Rodrigues, C. Honorato, J. Lourenço *et al.*, "Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation." in *HotDep*, 2008.

[194] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis *et al.*, "Zeno: Eventually consistent byzantine-fault tolerance." in *NSDI*, vol. 9, 2009, pp. 169–184.

[195] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[196] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman, "Eventually-serializable data services," *Theoretical Computer Science*, vol. 220, no. 1, pp. 113–156, 1999.

[197] B. Wester, J. A. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov, "Tolerating latency in replicated state machines through client speculation." in *NSDI*, 2009, pp. 245–260.

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

[199] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[200] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 135–144.

[201] C.-S. Barcelona, "Mencius: building efficient replicated state machines for wans," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[202] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter, "Zzyzx: Scalable fault tolerance through byzantine locking," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 363–372.

[203] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in *Proceedings of the thirteenth ACM symposium on Operating systems principles*, 1991, pp. 198–212.

[204] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads." in *USENIX annual technical conference*, vol. 1, no. 2, 2008, pp. 5–2.

[205] O. Rütti, Z. Milosevic, and A. Schiper, "Generic construction of consensus algorithms for benign and byzantine faults," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 343–352.

[206] A. Mostéfaoui, S. Rajsbaum, and M. Raynal, *A versatile and modular consensus protocol*. IRISA, 2001.

[207] R. Guerraoui and M. Raynal, "The information structure of indulgent consensus," *IEEE Transactions on Computers*, vol. 53, no. 4, pp. 453–466, 2004.

[208] Y. J. Song, R. Van Renesse, F. B. Schneider, and D. Dolev, "The building blocks of consensus," in *International Conference on Distributed Computing and Networking*. Springer, 2008, pp. 54–72.

[209] V. King and J. Saia, "Breaking the o (n 2) bit barrier: scalable byzantine agreement with an adaptive adversary," *Journal of the ACM (JACM)*, vol. 58, no. 4, pp. 1–24, 2011.

[210] L. Lamport, "Byzantizing paxos by refinement," in *International Symposium on Distributed Computing*. Springer, 2011, pp. 211–224.

[211] B. Lampson, "The abcd's of paxos," in *PODC*, vol. 1. Citeseer, 2001, p. 13.

[212] L. Lamport, *Specifying systems*. Addison-Wesley Boston, 2002, vol. 388.

[213] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.

[214] A. Shoker, M. Yabandeh, R. Guerraoui, and J.-P. Bahsoun, "Obfuscated bft," 2012.

[215] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "Zz and the art of practical bft execution," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 123–138.

[216] T. Distler and R. Kapitza, "Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 91–106.

[217] A. N. Bessani and M. Santos, "Bft-smart-high-performance byzantine-faulttolerant state machine replication," 2011.

[218] J. J. Stephen and P. Eugster, "Assured cloud-based data analysis with clusterbft," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2013, pp. 82–102.

[219] A. Shoker and J.-P. Bahsoun, "Bft selection," in *International Conference on Networked Systems*. Springer, 2013, pp. 258–262.

[220] C. Copeland and H. Zhong, "Tangaroa: a byzantine fault tolerant raft," 2016.

[221] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *International Conference on Principles of Distributed Systems*. Springer, 2014, pp. 91–106.

[222] S. Duan, K. Levitt, H. Meling, S. Peisert, and H. Zhang, "Byzid: Byzantine fault tolerance from intrusion detection," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE, 2014, pp. 253–264.

[223] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on software engineering*, no. 2, pp. 222–232, 1987.

[224] T. F. Lunt and R. Jagannathan, "A prototype real-time intrusion-detection expert system." in *IEEE Symposium on Security and Privacy*, vol. 59. Oakland, CA, USA, 1988.

[225] C. Ko, M. Ruschitzka, and K. Levitt, "Execution monitoring of security-critical programs in distributed systems: A specification-based approach," in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*. IEEE, 1997, pp. 175–187.

[226] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[227] A. S. de Sá, A. E. Silva Freitas, and R. J. de Araújo Macêdo, "Adaptive request batching for byzantine replication," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 35–42, 2013.

[228] S. Duan, S. Peisert, and K. N. Levitt, "hbft: speculative byzantine fault tolerance with minimum cost," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 1, pp. 58–70, 2014.

[229] J. Behl, T. Distler, and R. Kapitza, "Consensus-oriented parallelization: How to earn your first million," in *Proceedings of the 16th Annual Middleware Conference*, 2015, pp. 173–184.

[230] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 17–30.

[231] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 251–260.

[232] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE transactions on computers*, vol. 52, no. 2, pp. 139–149, 2003.

[233] G. Wang and M. Nixon, "Randchain: Practical scalable decentralized randomness attested by blockchain," in *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2020, pp. 442–449.

[234] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues, "Visigoth fault tolerance," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–14.

[235] M. E. B. Pires, "Generalized paxos made byzantine, visigoth and less complex," 2017.

[236] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "Sarek: Optimistic parallel ordering in byzantine fault tolerance," in *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016, pp. 77–88.

[237] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[238] J. Kwon, "Tendermint: Consensus without mining," *Draft v. 0.6, fall*, vol. 1, no. 11, 2014.

[239] G. Danezis and S. Meiklejohn, "Centrally banked cryptocurrencies," *arXiv preprint arXiv:1505.06895*, 2015.

[240] B. Laurie, "An efficient distributed currency," *Practice*, vol. 100, 2011.

[241] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th {usenix} security symposium ({usenix} security 16)*, 2016, pp. 279–296.

[242] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *2016 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2016, pp. 526–545.

[243] S. Karkhanis, "Evaluation of using pairing-based cryptography in byzcoin," 2019.

[244] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, 2016, pp. 45–59.

[245] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[246] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient byzantine fault tolerance," *IEEE transactions on computers*, vol. 65, no. 9, pp. 2807–2819, 2015.

[247] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solida: A blockchain protocol based on reconfigurable byzantine consensus," *arXiv preprint arXiv:1612.02916*, 2016.

[248] R. Pass and E. Shi, "Hybrid consensus: Efficient consensus in the permissionless model," in *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[249] A. Biryukov, D. Feher, and D. Khovratovich, "Guru: Universal reputation module for distributed consensus protocols," University of Luxembourg, Tech. Rep., 2017.

[250] B. Awerbuch and C. Scheideler, "Towards a scalable and robust dht," *Theory of Computing Systems*, vol. 45, no. 2, pp. 234–260, 2009.

[251] S. Sen and M. J. Freedman, "Commensal cuckoo: Secure group partitioning for large-scale services," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 33–39, 2012.

[252] L. Ren, K. Nayak, I. Abraham, and S. Devadas, "Practical synchronous byzantine consensus," *arXiv preprint arXiv:1704.02397*, 2017.

[253] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[254] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, "Troxy: Transparent access to byzantine fault-tolerant systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 59–70.

[255] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 3–33.

[256] ——, "Thunder research rafael pass and elaine shi august 29, 2018," 2018.

[257] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer, "Bounds on the time to reach agreement in the presence of timing uncertainty," *Journal of the ACM (JACM)*, vol. 41, no. 1, pp. 122–152, 1994.

[258] N. Shrestha, I. Abraham, L. Ren, and K. Nayak, "On the optimality of optimistic responsiveness," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 839–857.

[259] A. Momose, J. P. Cruz, and Y. Kaji, "Hybrid-bft: Optimistically responsive synchronous consensus with optimal latency or resilience." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 406, 2020.

[260] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Optimal good-case latency for byzantine broadcast and state machine replication," *arXiv preprint arXiv:2003.13155*, 2020.

[261] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 106–118.

[262] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," *arXiv preprint arXiv:1708.03778*, 2017.

[263] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[264] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[265] C. Carvalho, D. Porto, L. Rodrigues, M. Bravo, and A. Bessani, "Dynamic adaptation of byzantine consensus protocols," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 411–418.

[266] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.

[267] M. Bravo, L. Rodrigues, R. Neiheiser, and L. Rech, "Policy-based adaptation of a byzantine fault tolerant distributed graph database," in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2018, pp. 61–71.

[268] R. Neiheiser, D. Presser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia, "Fireplug: Flexible and robust n-version geo-replication of graph databases," in *2018 International Conference on Information Networking (ICOIN)*. IEEE, 2018, pp. 110–115.

[269] I. Abraham, T. H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, and E. Shi, "Communication complexity of byzantine agreement, revisited," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 317–326.

[270] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected o (1) rounds, expected

$$o(n\Theta 2)$$

communication, and optimal resilience," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 320–334.

[271] J. Katz and C.-Y. Koo, "On expected constant-round protocols for byzantine agreement," in *Annual International Cryptology Conference*. Springer, 2006, pp. 445–462.

[272] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, D.-A. Seredinschi, and Y. Vonlanthen, "Scalable byzantine reliable broadcast (extended version)," *arXiv preprint arXiv:1908.01738*, 2019.

[273] R. Cohen, I. Haitner, N. Makriyannis, M. Orland, and A. Samorodnitsky, "On the round complexity of randomized byzantine agreement," *arXiv preprint arXiv:1907.11329*, 2019.

[274] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International conference on the theory and application of cryptology and information security*. Springer, 2001, pp. 514–532.

[275] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1041–1053.

[276] S. Basu, A. Tomescu, I. Abraham, D. Malkhi, M. K. Reiter, and E. G. Sirer, "Efficient verifiable secret sharing with share recovery in bft protocols," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2387–2402.

[277] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.

[278] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual international cryptology conference*. Springer, 1991, pp. 129–140.

[279] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and secure global payments with stellar," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 80–96.

[280] G. Voron and V. Gramoli, "Dispel: byzantine smr with distributed pipelining," *arXiv preprint arXiv:1912.10367*, 2019.

[281] M. Scharf and S. Kiesel, "Nxg03-5: Head-of-line blocking in tcp and sctp: Analysis and measurements," in *IEEE Globecom 2006*. IEEE, 2006, pp. 1–5.

[282] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "Proof-of-

execution: Reaching consensus through fault-tolerant speculation," *arXiv preprint arXiv:1911.00838*, 2019.

[283] S. Bano, M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman *et al.*, "State machine replication in the libra blockchain," *Avalaible at: https://developers. libra. org/docs/state-machine-replication-paper (Consulted on December 19, 2020)*, 2020.

[284] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[285] A. Spiegelman, "In search for a linear byzantine agreement," *arXiv preprint arXiv:2002.06993*, 2020.

[286] J. Mickens, "The saddest moment," *Login Usenix Mag*, vol. 39, no. 3, pp. 52–54, 2014.

[287] M. M. Jalalzai, J. Niu, and C. Feng, "Fast-hotstuff: A fast and resilient hotstuff protocol," *arXiv preprint arXiv:2010.11454*, 2020.

[288] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 633–649.

[289] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "Resilientdb: global scale resilient blockchain fabric," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 868–883, 2020.

[290] G. Association *et al.*, "Blockchain for development: Emerging opportunities for mobile, identity and aid," 2017.

[291] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2008.

[292] J. Sousa and A. Bessani, "Separating the wheat from the chaff: An empirical design for geo-replicated state machines," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2015, pp. 146–155.

[293] C. Berger, H. P. Reiser, J. Sousa, and A. N. Bessani, "Aware: Adaptive wide-area replication for fast and resilient byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[294] Z. Avarikioti, L. Heimbach, R. Schmid, and R. Wattenhofer, "Fnf-bft: Exploring performance limits of bft protocols," *arXiv preprint arXiv:2009.02235*, 2020.

[295] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-glove approach for bft testing," *arXiv preprint arXiv:2004.10617*, 2020.

[296] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Gemini: Bft systems made robust."

[297] A. Stewart and E. Kokoris-Kogia, "Grandpa: a byzantine finality gadget," *arXiv preprint arXiv:2007.01560*, 2020.

[298] Z. Xiang, D. Malkhi, K. Nayak, and L. Ren, "Strengthened fault tolerance in byzantine fault tolerant replication," *arXiv preprint arXiv:2101.03715*, 2021.

[299] M. Platania, D. Obenshain, T. Tantillo, Y. Amir, and N. Suri, "On choosing server-or client-side solutions for bft," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–30, 2016.

[300] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 237–250.

[301] M. Eischer and T. Distler, "Latency-aware leader selection for geo-replicated byzantine fault-tolerant systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 140–145.

[302] R. Friedman and R. Van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)*. IEEE, 1997, pp. 233–242.

[303] H. P. Reiser and R. Kapitza, "Hypervisor-based efficient proactive recovery," in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 2007, pp. 83–92.

[304] T. Distler, I. Popov, W. Schröder-Preikschat, H. P. Reiser, and R. Kapitza, "Spare: Replicas on hold." in *NDSS*, 2011.

[305] M. Eischer and T. Distler, "Scalable byzantine fault-tolerant state-machine replication on heterogeneous servers," *Computing*, vol. 101, no. 2, pp. 97–118, 2019.

[306] G. Habiger, F. J. Hauck, J. Köstler, and H. P. Reiser, "Resource-efficient¨ state-machine replication with multithreading and vertical scaling," in *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 2018, pp. 87–94.

[307] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 575–584.

[308] M. Eischer, M. Büttner, and T. Distler, "Deterministic fuzzy checkpoints," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 153–15 309.

[309] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 452–465, 2009.

[310] P. Kuznetsov and R. Rodrigues, "Bftw3: Why? when? where? workshop on the theory and practice of byzantine fault tolerance," *ACM SIGACT News*, vol. 40, no. 4, pp. 82–86, 2010.

[311] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual international cryptology conference*. Springer, 1992, pp. 139–147.

[312] A. Back, "A partial hash collision based postage scheme," *Retrieved December*, vol. 29, p. 2018, 1997.

[313] I. Bentov, R. Pass, and E. Shi, "Snow white: Provably secure proofs of stake." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 919, 2016.

[314] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*. Springer, 2017, pp. 357–388.

[315] B. David, P. Gaži, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 66–98.

[316] A. Chepurnoy, "Interactive proof-of-stake," *arXiv preprint arXiv:1601.00275*, 2016.

[317] D. Larimer, "Delegated proof-of-stake (dpos). bitshare whitepaper (2014)," 2014.

[318] K. Karantias, A. Kiayias, and D. Zindros, "Proof-of-burn," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 523–540.

[319] T. NEM, "Nem technical reference," *URL https://nem. io/wpcontent/themes/nem/files/NEM_techRef. pdf*, 2018.

[320] V. Dhillon, D. Metcalf, and M. Hooper, "The hyperledger project," in *Blockchain enabled applications*. Springer, 2017, pp. 139–149.

[321] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz, "Permacoin: Repurposing bitcoin work for data preservation," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 475–490.

[322] S. Park, A. Kwon, G. Fuchsbauer, P. Gaži, J. Alwen, and K. Pietrzak, "Spacemint: A cryptocurrency based on proofs of space," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 480–499.

[323] L. Luu, Y. Velner, J. Teutsch, and P. Saxena, "Smartpool: Practical decentralized pooled mining," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1409–1426.

[324] I. Abraham, D. Malkhi *et al.*, "The blockchain consensus layer and bft," *Bulletin of EATCS*, vol. 3, no. 123, 2017.

[325] V. Buterin, "Casper version 1 implementation guide," *Ethereum Github repository*, 2017.

[326] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus," *CoRR, abs/1612.02916*, 2016.

[327] V. King, J. Saia, V. Sanwalani, and E. Vee, "Scalable leader election," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, 2006, pp. 990–999.

[328] D. Dasgupta, J. M. Shrein, and K. D. Gupta, "A survey of blockchain from security perspective," *Journal of Banking and Financial Technology*, vol. 3, no. 1, pp. 1–17, 2019.

[329] S. M. H. Bamakan, A. Motavali, and A. B. Bondarti, "A survey of blockchain consensus algorithms performance evaluation criteria," *Expert Systems with Applications*, p. 113385, 2020.

[330] H. Hasanova, U.-j. Baek, M.-g. Shin, K. Cho, and M.-S. Kim, "A survey on blockchain cybersecurity vulnerabilities and possible countermeasures," *International Journal of Network Management*, vol. 29, no. 2, p. e2060, 2019.

[331] G. Bissias, B. N. Levine, A. P. Ozisik, and G. Andresen, "An analysis of attacks on blockchain consensus," *arXiv preprint arXiv:1610.07985*, 2016.

[332] S. Sayeed and H. Marco-Gisbert, "Assessing blockchain consensus and security mechanisms against the 51% attack," *Applied Sciences*, vol. 9, no. 9, p. 1788, 2019.

[333] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.

[334] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 7, pp. 1366–1385, 2018.

[335] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 3–16.

[336] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[337] J. Bruce, "The mini-blockchain scheme," *White paper*, 2014.

[338] G. Wang, Z. Shi, M. Nixon, and S. Han, "Chainsplitter: Towards blockchain-based industrial iot architecture for supporting hierarchical storage," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 166–175.

[339] A. Biryukov, D. Khovratovich, and I. Pustogarov, "Deanonymisation of clients in bitcoin p2p network," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 15–29.

[340] S. Feld, M. Schönfeld, and M. Werner, "Analyzing the deployment of bitcoin's p2p network under an as-level perspective," *Procedia Computer Science*, vol. 32, pp. 1121–1126, 2014.

[341] P. Koshy, D. Koshy, and P. McDaniel, "An analysis of anonymity in bitcoin using p2p network traffic," in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 469–485.

[342] A. Kumar, C. Fischer, S. Tople, and P. Saxena, "A traceability analysis of monero's blockchain," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 153–173.

[343] G. Maxwell, "Coinjoin: Bitcoin privacy for the real world," in *Post on Bitcoin forum*, 2013.

[344] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "Coinshuffle: Practical decentralized coin mixing for bitcoin," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 345–364.

[345] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "An in-depth look of bft consensus in blockchain: Challenges and opportunities," in *Proceedings of the 20th International Middleware Conference Tutorials*, 2019, pp. 6–10.

[346] H. Du and D. J. S. Hilaire, "Multi-paxos: An implementation and evaluation," *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02*, 2009.

[347] J. Nijsse and A. Litchfield, "A taxonomy of blockchain consensus methods," *Cryptography*, vol. 4, no. 4, p. 32, 2020.

[348] I. Sheff, X. Wang, R. van Renesse, and A. C. Myers, "Heterogeneous paxos," in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[349] P. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failure modes," in *Proceedings Seventh Symposium on Reliable Distributed Systems*. IEEE Computer Society, 1988, pp. 93–94.

[350] J. M. Rushby, "Design and verification of secure systems," *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5, pp. 12–21, 1981.

[351] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Ebawa: Efficient byzantine agreement for wide-area networks," in *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. IEEE, 2010, pp. 10–19.

[352] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM New York, NY, USA, 2013, p. 7.

[353] A. ARM, "Security technology-building a secure system using trustzone technology," *ARM Technical White Paper*, 2009.

[354] S. Rüsch, K. Bleeke, and R. Kapitza, "Bloxy: Providing transparent and generic bft-based ordering services for blockchains," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 305–30 509.

[355] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo, "Worm-it–a wormhole-based intrusion-tolerant group communication system," *Journal of Systems and Software*, vol. 80, no. 2, pp. 178–197, 2007.

[356] M. Garcia, A. Bessani, and N. Neves, "Lazarus: Automatic management of diversity in bft systems," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 241–254.

[357] M. Bravo, G. Chockler, and A. Gotsman, "Making byzantine consensus live," in *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[358] P. Tholoniat and V. Gramoli, "Formal verification of blockchain byzantine fault tolerance," *arXiv preprint arXiv:1909.07453*, 2019.

[359] D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, "Dynamic fault-tolerant clock synchronization," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 143–185, 1995.

[360] B. Simons, "An overview of clock synchronization," *Fault-Tolerant Distributed Computing*, pp. 84–96, 1990.

[361] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of bitcoins: characterizing payments among men with no names," in *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 127–140.

[362] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto, "Bigchaindb: a scalable blockchain database," *white paper, BigChainDB*, 2016.

[363] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2010, pp. 321–340.

[364] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: architecture, applications, and future trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 11, pp. 2266–2277, 2019.

[365] R. Zhang, R. Xue, and L. Liu, "Security and privacy on blockchain," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–34, 2019.

[366] K. Birman, "The promise, and limitations, of gossip protocols," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 8–13, 2007.

[367] A.-M. Kermarrec and M. Van Steen, "Gossiping in distributed systems," *ACM SIGOPS operating systems review*, vol. 41, no. 5, pp. 2–7, 2007.

[368] S. B. Mokhtar, A. Pace, and V. Quéma, "Firespam: Spam resilient gossiping in the bar model," in *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 225–234.

[369] L. Alvisi, J. Doumen, R. Guerraoui, B. Koldehofe, H. Li, R. Van Renesse, and G. Tredan, "How robust are gossip-based communication

protocols?" *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 14–18, 2007.

[370] R. Van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Middleware'98*. Springer, 1998, pp. 55–70.

[371] L. Gelbmann and G. Bosson, "Bls cosigning via a gossip protocol," 2019.

[372] J. F. Mikalsen, "Firechain: An efficient blockchain protocol using secure gossip," Master's thesis, UiT Norges arktiske universitet, 2018.

[373] R. van Renesse, "A blockchain based on gossip?-a position paper," *Cornell University*, 2016.

[374] M. Correia, N. F. Neves, and P. Verissimo, "Bft-to: Intrusion tolerance with less replicas," *The Computer Journal*, vol. 56, no. 6, pp. 693–715, 2012.

[375] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Efficient synchronous byzantine consensus," *arXiv preprint arXiv:1704.02397*, 2017.

[376] I. Grigg, "Eos-an introduction," *White paper. https://whitepaperdatabase. com/eos-whitepaper*, 2017.

[377] H. Dang, A. Dinh, E.-C. Chang, and B. C. Ooi, "Chain of trust: Can trusted hardware help scaling blockchains?" *arXiv preprint arXiv:1804.00399*, 2018.

[378] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[379] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 29–37, 2014.

[380] C. Stathakopoulous and C. Cachin, "Threshold signatures for blockchain systems," *Swiss Federal Institute of Technology*, 2017.

[381] S. Gupta, S. Rahnama, and M. Sadoghi, "Permissioned blockchain through the looking glass: Architectural and implementation lessons learned," *arXiv preprint arXiv:1911.09208*, 2019.

[382] S. Gupta and M. Sadoghi, "Easycommit: A non-blocking two-phase commit protocol." in *EDBT*, 2018, pp. 157–168.

[383] T. M. Qadah and M. Sadoghi, "Quecc: A queue-oriented, control-free concurrency architecture," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 13–25.

[384] T. Qadah, S. Gupta, and M. Sadoghi, "Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication." in *EDBT*, 2020, pp. 73–84.

[385] S. Gupta, "Resilient and scalable architecture for permissioned blockchain fabrics," *Proceedings of the VLDB Endowment*, 2020.

[386] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 368–377.

[387] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[388] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "Caper: a cross-application permissioned blockchain," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1385–1398, 2019.

[389] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 123–140.

[390] N. Szabo, "The idea of smart contracts," *Nick Szabo's papers and concise tutorials*, vol. 6, no. 1, 1997.

[391] C. Jentzsch, "The history of the dao and lessons learned," *Slock. it Blog*, vol. 24, 2016.

[392] H. Wan, K. Li, and Y. Huang, "Blockchain: A review from the perspective of operations researchers," in *2020 Winter Simulation Conference (WSC)*. IEEE, 2020, pp. 75–89.