Flexible Byzantine Fault Tolerance

Dahlia Malkhi[1], Kartik Nayak[1,2], and Ling Ren[1,3]

[1]VMware Research – {dmalkhi,nkartik,lingren}@vmware.com
[2]Duke University
[3]University of Illinois at Urbana-Champaign

April 29, 2019

Abstract

Existing Byzantine fault tolerant (BFT) protocols work in a homogeneous model where a service administrator picks a set of assumptions (timing model and the fraction of Byzantine faults) and imposes these assumptions on all clients using the service. This paper introduces Flexible BFT, a family of BFT protocols that support clients with heterogenous assumptions. In a Flexible BFT protocol, replicas execute a set of instructions while each client decides whether a transaction is committed based on its own assumption. At a technical level, Flexible BFT makes two key contributions. First, it introduces a synchronous BFT protocol in which only the commit step requires to know the network delay bound and thus replicas execute the protocol without any synchrony assumption. Second, it introduces a notion called Flexible Byzantine Quorums by deconstructing the roles of different quorums in existing consensus protocols. This paper also introduces a new type of fault called *alive-but-corrupt* faults: adversaries that attack safety but maintain liveness. Flexible BFT can tolerate a combination of Byzantine and alive-but-corrupt faults that exceed one-third with partial synchrony or exceeds one-half with synchrony while still respecting Byzantine fault tolerance bounds in respective models.

1 Introduction

Byzantine fault tolerant (BFT) protocols are used to build replicated services. Recently, they have received revived interest as the algorithmic foundation of what are known as decentralized ledgers, or blockchains.

In the classic approach to BFT protocol designs, a protocol designer or a service administrator first picks a set of assumptions (e.g., the fraction of Byzantine faults and certain timing assumptions) and then devises a protocol (or chooses an existing one) tailored for that particular setting. The assumptions made by the protocol designer are imposed upon all parties involved — every replica maintaining the service as well as every client using the service. Such a protocol collapses if deployed under settings that differ from the one it is designed for. In particular, optimal-resilience partially synchronous solutions [10, 9] completely break if the fraction of Byzantine faults exceeds 1/3. Similarly, optimal-resilience synchronous solutions [1, 13] break if the fraction of Byzantine faults exceeds 1/2 or if the synchrony bound is violated.

In this work, we introduce a new approach for BFT protocol design called *Flexible BFT*. Our approach offers advantages in the two aspects above. First, the Flexible BFT approach enables protocols that tolerate more than 1/3 (resp. 1/2) faults in the partial-synchrony (resp. synchrony) model — not all faults are Byzantine, in accordance with the above resilience bounds. Second, the Flexible BFT approach allows a certain degree of separation between the fault model and the protocol design. As a result, Flexible BFT allows *heterogeneous* clients with different fault assumptions and timing assumptions (synchrony or not) to participate in the same protocol. We elaborate on these two aspects below.

Beyond resilience bounds. In order to bypass known resilience bounds, we introduce a mixed fault model with a new type of fault called *alive-but-corrupt* faults. An alive-but-corrupt replica tries to violate safety but not liveness. The rationale for this new type of fault is that violating safety may provide the

attacker gains (e.g., a double spend attack) but preventing liveness usually does not. In fact, alive-but-corrupt replicas may gain rewards from keeping the replicated service live, e.g., by collecting service fees. We show a family of protocols that tolerate a combination of Byzantine and alive-but-corrupt faults that exceeds 1/3 in the partially synchronous model and exceeds 1/2 in the synchronous model. Our results do not violate existing resilience bounds because the fraction of Byzantine faults is always smaller than the respective bounds.

One consensus protocol for the populace. The Flexible BFT approach further provides certain separation between the fault model and the protocol and allows clients with different assumptions to co-exist in the same protocol. Clients specify (i) the fault threshold they need to tolerate, and (ii) the message delay bound, if any, they believe in. For example, one instance of Flexible BFT can support a client that requires tolerance against 1/5 Byzantine faults plus 3/10 alive-but-corrupt faults, while simultaneously supporting another client who requires tolerance against 1/10 Byzantine faults plus 1/2 alive-but-corrupt faults, and a third client who believes in synchrony and requires 3/10 Byzantine plus 2/5 alive-but-corrupt tolerance.

This novel separation of fault model from protocol design can be useful in practice in several ways. Different clients may naturally hold different assumptions about the system. Some clients may be more cautious and require a higher resilience than others; some clients may believe in synchrony while others do not. Moreover, even the same client may assume a larger fraction of faults when dealing with a $1M transaction compared to a $5 one. The rationale is that more replicas may be willing to collude to double spend a high-value transaction. In addition, if a client observes votes to conflicting values, which indicates the system is under attack, the client can choose to be more cautious and require higher resilience than usual.

The notion of "commit" needs to be clarified in our new model. Clients in Flexible BFT have different assumptions and hence different commit rules. It is then possible and common that a value is committed by one client but not another. Flexible BFT guarantees that any two clients whose assumptions are correct (but possibly different) commit to the same value. If a client's assumption is incorrect, however, it may commit inconsistent values which may later be reverted. While this new notion of commit may sound radical at first, it is the implicit behavior of existing BFT protocols. If the assumption made by the service administrator is violated in a classic BFT protocol (e.g., there are more Byzantine faults than provisioned), clients may commit to different values and they have no recourse. In this sense, Flexible BFT is a robust generalization of classic BFT protocols. In Flexible BFT, if a client performs conflicting commits, it should update its assumption to be more cautious and re-interpret what values are committed under its new assumption. In fact, this "recovery" behavior is somewhat akin to Bitcoin. A client in Bitcoin decides how many confirmations are needed (i.e., how "deeply buried") to commit a block. If the client commits but subsequently an alternative longer fork appears, its commit is reverted. Going forward, the client may increase the number of confirmations it requires.

Key techniques. Flexible BFT centers around two new techniques. The first one is a novel synchronous BFT protocol with replicas executing at *network speed*; that is, the protocol run by the replicas does not assume synchrony. This allows clients in the same protocol to assume different message delay bounds and commit at their own pace. The protocol thus separates timing assumptions of replicas from timing assumptions of clients. Note that this is only possible via Flexible BFT's separation of protocol from fault model: the action of committing is only carried out by clients, not by replicas. The other technique involves a breakdown of the different roles that quorums play in different steps of partially synchronous BFT protocols. Once again, made possible by the separation in Flexible BFT, we will use one quorum size for replicas to run a protocol, and let clients choose their own quorum sizes for committing in the protocol.

Contributions. To summarize, our work has the following contributions.

1. **Synchronous BFT with network speed replicas.** We present a synchronous protocol in which only the commit step requires synchrony. Since replicas no longer commit in our approach, the protocol simultaneously supports clients assuming different synchrony bounds.

2. **Flexible Byzantine Quorums.** We deconstruct existing BFT protocols to understand the role played by different quorums and introduce the notion of Flexible Byzantine Quorums. A protocol based on Flexible Byzantine Quorums simultaneously supports clients assuming different fault models.

3. **Flexible BFT.** Putting these together, we present a new approach for BFT design, Flexible BFT. Our approach supports clients with varying fault and timing assumptions in the same protocol and tolerates a fraction of combined (Byzantine plus alive-but-corrupt) faults beyond existing resilience bounds.

Organization. The rest of the paper is organized as follows. Section 2 defines the Flexible BFT model where replicas and clients are separated. We will describe in more detail our key techniques for synchrony and partial-synchrony in Sections 3 and 4, respectively. Section 5 puts these techniques together and presents the final protocol. Section 6 discusses the result obtained by the Flexible BFT design and Section 7 describes related work.

2 Modeling Flexible BFT

The goal of Flexible BFT is to build a replicated service that takes requests from clients and provides clients an interface of a single non-faulty server, i.e., it provides clients with the same totally ordered sequence of values. Internally, the replicated service uses multiple servers, also called replicas, to tolerate some number of faulty servers. The total number of replicas is denoted by n. In this paper, whenever we speak about a set of replicas or messages, we denote the set size as its fraction over n. For example, we refer to a set of m replicas as "q replicas" where $q = m/n$.

Borrowing notation from Lamport [17], such a replicated service has three logical actors: *proposers* capable of sending new values, *acceptors* who add these values to a totally ordered sequence (called a blockchain), and *learners* who decide on a sequence of values based on the transcript of the protocol and execute them on a state machine. Existing replication protocols provide the following two properties:

(Safety) Any two learners learn the same sequence of values.

(Liveness) A value proposed by a proposer will eventually be executed by every learner.

In existing replication protocols, the learners are assumed to be homogeneous, i.e., they interpret a transcript using the same rules and hence decide on the same sequence of values. In Flexible BFT, we consider heterogeneous learners with different assumptions. Based on their own assumptions, they may interpret the transcript of the protocol differently. We show that so far as the assumptions of two different learners are both correct, they will eventually learn the same sequence of values. A replication protocol in the Flexible BFT approach satisfies the following properties:

(Safety for heterogeneous learners) Any two learners with correct but potentially different assumptions learn the same sequence of values.

(Liveness for heterogeneous learners) A value proposed by a proposer will eventually be executed by every learner with a correct assumption.

In a replicated service, clients act as proposers and learners, whereas the replicas (replicated servers) are acceptors. Thus, safety and liveness guarantees are defined with respect to the heterogeneous clients.

Fault model. We assume two types of faults within the replicas: Byzantine and *alive-but-corrupt*. Byzantine replicas behave arbitrarily. On the other hand, the goal of alive-but-corrupt replicas is to attack safety but to preserve liveness. We assume that the adversary is static, i.e., the adversary determines which replicas are Byzantine and alive-but-corrupt before the start of the protocol. We remark that with this new fault model, the safety proof should treat alive-but-corrupt replicas similarly to Byzantine. Then, *once safety is proved*, the liveness proof can treat alive-but-corrupt replicas similarly to honest.

Other assumptions. We assume hash functions, digital signatures and a public-key infrastructure (PKI). We use $\langle x \rangle_R$ to denote a message x signed by a replica R. We assume pair-wise communication channels between replicas. We assume that all replicas have clocks that advance at the same rate.

3 Synchronous BFT with Network Speed Replicas - Overview

Early synchronous protocols have relied on synchrony in two ways. First, the replicas assume a maximum network delay Δ for communication between them. Second, they require a lock step execution, i.e., all replicas are in the same round at the same time. Hanke et al. showed a synchronous protocol without lock step execution [13]. Their protocol still contains a synchronous step in which all replicas perform a blocking wait of 2Δ time before proceeding to subsequent steps. Sync HotStuff [4] improves on it further to remove replicas' blocking waits during good periods (when the leader is honest), but blocking waits are still required by replicas during bad situations (view changes).

In this section, we show a synchronous protocol where the replicas execute at the network speed. In other words, replicas run a partially synchronous protocol and do not rely on synchrony at any point. Clients, on the other hand, rely on synchrony bounds to commit. This separation is what allows our protocol to support clients with different assumptions on the value of Δ. To the best of our knowledge, this is the first synchronous protocol to achieve such a separation. In addition, the protocol tolerates a combined Byzantine plus alive-but-corrupt fault ratio greater than a half (Byzantine fault tolerance is still less than half).

For simplicity, in this overview, we show a protocol for single shot consensus. In our final protocol in Section 5, we will consider a pipelined version of the protocol for consensus on a sequence of values. We do not consider termination for the single-shot consensus protocol in this overview because our final replication protocol is supposed to run forever.

The protocol is shown in Figure 1. It runs in a sequence of views. Each view has a designated leader who may be selected in a round robin order. The leader drives consensus in that view. In each view, the protocol runs in two steps – propose and vote. In the propose step, the leader proposes a value b. In the vote step, replicas vote for the value if it is *safe* to do so. The vote also acts as a *re-proposal* of the value. If a replica observes a set of q_r votes on b, called a certificate $\mathcal{C}^{q_r}(b)$, it "locks" on b. Looking ahead, we will observe that $q_r > 1/2$; we will revisit the choice of q_r for Flexible BFT in Section 6. In subsequent views, a replica will not vote for a value other than b unless it learns that q_r replicas are not locked on b. In addition, the replicas switch views (i.e., change leader) if they either observe an equivocation or if they do not receive a proposal from the leader within some timeout. A client commits b if q_r replicas state that there exists a view in which b is certified and no equivocating value or view change was observed at a time before 2Δ after it was certified. Here, Δ represents is the maximum network delay the client believes in.

The protocol ensures safety if there are fewer than q_r faulty replicas. The key argument for safety is the following: If an honest replica h satisfies the commit condition for some value b in a view, then (a) no other value can be certified and (b) all honest replicas are locked on b at the end of that view. To elaborate, satisfying the commit condition implies that some honest replica h has observed an undisturbed-2Δ period after it locked on b, i.e., it did not observe an equivocation or a view change. Suppose the condition is satisfied at time t. This implies that other replicas did not observe an equivocation or a view change before $t-\Delta$. The two properties above hold if the quorum honesty conditions below hold. For liveness, if Byzantine leaders equivocate or do not propose a safe value, they will be blamed and a view change will ensue. Eventually there will be an honest leader to drive consensus if quorum availability holds.

Quorum honesty (a) within a view. Since the undisturbed period starts after b is certified, h must have voted (and re-proposed) b at a time earlier than $t - 2\Delta$. Every honest replica must have received b before $t - \Delta$. Since they had not voted for an equivocating value by then, they must have voted for b. Since the number of faults is less than q_r, every certificate needs to contain an honest replica's vote. Thus, no certificate for any other value can be formed in this view.

Quorum honesty (b) across views. h sends $\mathcal{C}_v^{q_r}(b)$ at time $t - 2\Delta$. All honest receive $\mathcal{C}_v^{q_r}(b)$ by time $t - \Delta$ and become locked on b. For an honest replica to unlock from b in subsequent views, q_r replicas need to claim that they are not locked on b. At least one of them is honest and would need to falsely claim it is not locked, which cannot happen.

Protocol executed by the replicas.

1. **Propose.** The leader L of view v proposes a value b.

2. **Vote.** On receiving the first value b in a view v, a replica broadcasts b and votes for b if it is *safe* to do so, as determined by a locking mechanism described later. The replica records the following.

   - If the replica collects q_r votes on b, denoted as $\mathcal{C}_v^{q_r}(b)$ and called a certificate of b from view v, then it "locks" on b and records the lock time as $\mathsf{t\text{-}lock}_v$.
   - If the replica observes an equivocating value signed by L at any time after entering view v, it records the time of equivocation as $\mathsf{t\text{-}equiv}_v$. It blames the leader by broadcasting $\langle \mathsf{blame}, v \rangle$ and the equivocating values.
   - If the replica does not receive a proposal for sufficient time in view v, it times out and broadcasts $\langle \mathsf{blame}, v \rangle$.
   - If the replica collects a set of q_r $\langle \mathsf{blame}, v \rangle$ messages, it records the time as $\mathsf{t\text{-}viewchange}_v$, broadcasts them and enters view $v + 1$.

If a replica locks on a value b in a view, then it votes only for b in subsequent views unless it "unlocks" from b by learning that q_r replicas are not locked on b in that view or higher (they may be locked on other values or they may not be locked at all).

Commit rules for clients. A value b is said to be committed by a client assuming Δ-synchrony iff q_r replicas each report that there exists a view v such that,

1. b is certified, i.e., $\mathcal{C}_v^{q_r}(b)$ exists.

2. b is undisturbed, i.e., no equivocating value or view change was observed at a time before 2Δ after it was certified, or more formally, $\min(\mathsf{current\text{-}time}, \mathsf{t\text{-}equiv}_v, \mathsf{t\text{-}viewchange}_v) - \mathsf{t\text{-}lock}_v \geq 2\Delta$

Figure 1: Synchronous BFT with network speed replicas.

Quorum availability. Byzantine replicas do not exceed $1 - q_r$ so that q_r replicas respond to the leader.

Tolerating alive-but-corrupt faults. If we have only honest and Byzantine replicas (and no alive-but-corrupt replicas), quorum honesty requires the fraction of Byzantine replicas $B < q_r$. Quorum availability requires $B \leq 1 - q_r$. If we optimize for maximizing B, we obtain $q_r \geq 1/2$. Now, suppose P represents the fraction of alive-but-corrupt replicas. Quorum honesty requires $B + P < q_r$, and quorum availability requires $B \leq 1 - q_r$. Thus, the protocol supports varying values of B and P at different values of $q_r > 1/2$ such that safety and liveness are both preserved.

Separating client synchrony assumption from the replica protocol. The most interesting aspect of this protocol is the separation of the client commit rule from the protocol design. In particular, although this is a synchronous protocol, the replica protocol does not rely on any synchrony bound. This allows clients to choose their own message delay bounds. Any client that uses a correct message delay bound enjoys safety.

4 Flexible Byzantine Quorums for Partial Synchrony - Overview

In this section, we explain the high-level insights of Flexible Byzantine Quorums in Flexible BFT. Again, for ease of exposition, we focus on a single-shot consensus and do not consider termination. We start by reviewing the Byzantine Quorum Systems [20] that underlie existing partially synchronous protocols that tolerate 1/3 Byzantine faults (Section 4.1). We will illustrate that multiple uses of 2/3-quorums actually serve different purposes in these protocols. We then generalize these protocols to use *Flexible Byzantine*

Quorums (Section 4.2), the key idea that enables more than 1/3 fault tolerance and allows heterogeneous clients with varying assumptions to co-exist.

4.1 Background: Quorums in PBFT

Existing protocols for solving consensus in the partially synchronous setting with optimal 1/3-resilience revolve around voting by *Byzantine quorums* of replicas. Two properties of Byzantine quorums are utilized for achieving safety and liveness. First, any two quorums intersect at one honest replica – quorum intersection. Second, there exists a quorum that contains no Byzantine faulty replicas – quorum availability. Concretely, when less than 1/3 the replicas are Byzantine, quorums are set to size $q_r = 2/3$. (To be precise, q_r is slightly larger than 2/3, i.e., $2f+1$ out of $3f+1$ where f is the number of faults, but we will use $q_r = 2/3$ for ease of exposition.) This guarantees an intersection of size at least $2q_r - 1 = 1/3$, hence at least one honest replica in the intersection. As for availability, there exist $q_r = 2/3$ honest replicas to form a quorum.

To dissect the use of quorums in BFT protocols, consider their use in PBFT [9] for providing safety and liveness. PBFT operates in a view-by-view manner. Each view has a unique leader and consists of the following steps:

- **Propose.** A leader L proposes a value b.
- **Vote 1.** On receiving the first value b for a view v, a replica votes for b if it is *safe*, as determined by a locking mechanism described below. A set of q_r votes form a certificate $\mathcal{C}^{q_r}(b)$.
- **Vote 2.** On collecting $\mathcal{C}^{q_r}(b)$, a replica "locks" on b and votes for $\mathcal{C}^{q_r}(b)$.
- **Commit.** On collecting q_r votes for $\mathcal{C}^{q_r}(b)$, a client learns that proposal b becomes a committed decision.

If a replica locks on a value b in a view, then it votes only for b in subsequent views unless it "unlocks" from b. A replica "unlocks" from b if it learns that q_r replicas are *not* locked on b in that view or higher (they may be locked on other values or they may not be locked at all).

The properties of Byzantine quorums are harnessed in PBFT for safety and liveness as follows:

Quorum intersection within a view. Safety within a view is ensured by the first round of votes. A replica votes only once per view. For two distinct values to both obtain certificates, one honest replica needs to vote for both, which cannot happen.

Quorum intersection across views. Safety across views is ensured by the locking mechanism. If b becomes a committed decision in a view, then a quorum of replicas lock on b in that view. For an honest replica among them to unlock from b, a quorum of replicas need to claim they are not locked on b. At least one replica in the intersection is honest and would need to falsely claim it is not locked, which cannot happen.

Quorum availability within a view. Liveness within each view is guaranteed by having an honest quorum respond to a non-faulty leader.

4.2 Flexible Byzantine Quorums

Our Flexible BFT approach separates the quorums used in BFT protocols for the replicas (acceptors) from the quorums used for learning when a decision becomes committed. More specifically, we denote the quorum used for forming certificates (locking) by q_{lck} and the quorum used for unlocking by q_{ulck}. We denote the quorum employed by clients for learning certificate uniqueness by q_{unq}, and the quorum used for learning commit safety by q_{cmt}. In other words, clients mandate q_{unq} first-round votes and q_{cmt} second-round votes in order to commit a decision. Below, we outline a modified PBFT-like protocol that uses these different quorum sizes instead of a single quorum size q. We then introduce a new definition, Flexible Byzantine Quorums, that capture the requirements needed for these quorums to provide safety and liveness.

- **Propose.** A leader L proposes a value b.
- **Vote 1.** On receiving the first value b for a view v, a replica votes for b if it is *safe*, as determined by a locking mechanism described below. A set of q_{lck} votes forms a certificate $\mathcal{C}^{q_{\text{lck}}}(b)$.
- **Vote 2.** On collecting $\mathcal{C}^{q_{\text{lck}}}(b)$, a replica "locks" on b and votes for $\mathcal{C}^{q_{\text{lck}}}(b)$.
- **Commit.** On collecting q_{unq} votes for b and q_{cmt} votes for $\mathcal{C}^{q_{\text{lck}}}(b)$, a client learns that proposal b becomes a committed decision.

If a replica locks on a value b in a view, then it votes only for b in subsequent views unless it "unlocks" from b by learning that q_{ulck} replicas are not locked on b.

Flexible quorum intersection (a) within a view. Contrary to PBFT, in Flexible BFT, a pair of q_{lck} certificates need not necessarily intersect in an honest replica. Indeed, locking on a value does not preclude conflicting locks. It only mandates that every q_{lck} quorum intersects with every q_{unq} quorum at at least one honest replica. For safety, it is essential that the fraction of faulty replicas is less than $q_{\text{lck}} + q_{\text{unq}} - 1$.

Flexible quorum intersection (b) across views. If a client commits a value b in a view, q_{cmt} replicas lock on b in that view. For an honest replica among them to unlock from b, q_{ulck} replicas need to claim they are not locked on b. This property mandates that every q_{ulck} quorum intersects with every q_{cmt} quorum at at least one honest replica. Thus, for safety, it is essential that the fraction of faulty replicas is less than $q_{\text{ulck}} + q_{\text{cmt}} - 1$.

Flexible quorum availability within each view. For liveness, Byzantine replicas cannot exceed $1 - \max(q_{\text{unq}}, q_{\text{cmt}}, q_{\text{lck}}, q_{\text{ulck}})$ so that the aforementioned quorums can be formed at different stages of the protocol.

Given the above analysis, Flexible BFT ensures safety if the fraction of faulty replicas is less than $\min(q_{\text{unq}} + q_{\text{lck}} - 1, q_{\text{cmt}} + q_{\text{ulck}} - 1)$, and provides liveness if the fraction of Byzantine replicas is at most $1 - \max(q_{\text{unq}}, q_{\text{cmt}}, q_{\text{lck}}, q_{\text{ulck}})$. It is optimal to use *balanced quorum sizes* where $q_{\text{lck}} = q_{\text{ulck}}$ and $q_{\text{unq}} = q_{\text{cmt}}$. To see this, first note that we should make sure $q_{\text{unq}} + q_{\text{lck}} = q_{\text{cmt}} + q_{\text{ulck}}$; otherwise, suppose the right-hand side is smaller, then setting $(q_{\text{cmt}}, q_{\text{ulck}})$ to equal $(q_{\text{unq}}, q_{\text{lck}})$ improves safety tolerance without affecting liveness tolerance. Next, observe that if we have $q_{\text{unq}} + q_{\text{lck}} = q_{\text{cmt}} + q_{\text{ulck}}$ but $q_{\text{lck}} > q_{\text{ulck}}$ (and hence $q_{\text{unq}} < q_{\text{cmt}}$), then once again setting $(q_{\text{cmt}}, q_{\text{ulck}})$ to equal $(q_{\text{unq}}, q_{\text{lck}})$ improves safety tolerance without affecting liveness tolerance.

Thus, in this paper, we set $q_{\text{lck}} = q_r$ and $q_{\text{unq}} = q_{\text{cmt}} = q_c$. Since replicas use q_r votes to lock, these votes can always be used by the clients to commit q_{cmt} quorums. Thus, $q_c \geq q_r$. The Flexible Byzantine Quorum requirements collapse into the following two conditions.

Flexible quorum intersection. The fraction of faulty replicas is $< q_c + q_r - 1$.

Flexible quorum availability. The fraction of Byzantine replicas is $\leq 1 - q_c$.

Tolerating alive-but-corrupt faults. If all faults in the system are Byzantine faults, then the best parameter choice is $q_c = q_r \geq 2/3$ for $< 1/3$ fault tolerance, and Flexible Byzantine Quorums degenerate to basic Byzantine quorums. However, in our model, alive-but-corrupt replicas are only interested in attacking safety but not liveness. This allows us to tolerate $q_c + q_r - 1$ total faults (Byzantine plus alive-but-corrupt), which can be more than 1/3. For example, if we set $q_r = 0.7$ and $q_c = 0.8$, then such a protocol can tolerate 0.2 Byzantine faults plus 0.3 alive-but-corrupt faults. We discuss the choice for q_r and q_c and their rationale in Section 6.

Separating client commit rules from the replica protocol. A key property of the Flexible Byzantine Quorum approach is that it decouples the BFT protocol from client commit rules. The decoupling allows

clients assuming different fault models to utilize the same protocol. In the above protocol, the propose and two voting steps are executed by the replicas and they are only parameterized by q_r. The commit step can be carried by different clients using different commit thresholds q_c. Thus, a fixed q_r determines a possible set of clients with varying commit rules (in terms of Byzantine and alive-but-corrupt adversaries). Recall that a Byzantine adversary can behave arbitrarily and thus may not provide liveness whereas a alive-but-corrupt adversary only intends to attack safety but not liveness. Thus, a client who believes that a large fraction of the adversary may attempt to break safety, not progress, can choose a larger q_c. By doing so, it seeks stronger safety against dishonest replicas, while trading liveness. Conversely, a client that assumes that a large fraction of the adversary attacks liveness must choose a smaller q_c.

5 Flexible BFT Protocol

In this section, we combine the ideas presented in Sections 3 and 4 to obtain a final protocol that supports both types of clients. A client can either assume partial synchrony, with freedom to choose q_c as described in the previous section, or assume synchrony with its own choice of Δ, as described in Section 3. Replicas execute a protocol at the network speed with a parameter q_r. We first give the protocol executed by the replicas and then discuss how clients commit depending on their assumptions. Moreover, inspired by Casper [7] and HotStuff [27], we show a protocol where the rounds of voting can be pipelined.

5.1 Notation

Before describing the protocol, we will first define some data structures and terminologies that will aid presentation.

Block format. The pipelined protocol forms a chain of values. We use the term *block* to refer to each value in the chain. We refer to a block's position in the chain as its *height*. A block B_k at height k has the following format

$$B_k := (b_k, h_{k-1})$$

where b_k denotes a proposed value at height k and $h_{k-1} := H(B_{k-1})$ is a hash digest of the predecessor block. The first block $B_1 = (b_1, \perp)$ has no predecessor. Every subsequent block B_k must specify a predecessor block B_{k-1} by including a hash of it. We say a block is *valid* if (i) its predecessor is valid or $\perp$, and (ii) its proposed value meets application-level validity conditions and is consistent with its chain of ancestors (e.g., does not double spend a transaction in one of its ancestor blocks). We say B_l *extends* B_k, if B_k is an ancestor of B_l ($l > k$). We say two blocks B_l and $B'_{l'}$ *equivocate* one another if they are not equal and do not extend one another.

Certificates and certified blocks. In the protocol, replicas vote for blocks by signing them. We use $\mathcal{C}_v^{q_r}(B_k)$ to denote a set of signatures on $h_k = H(B_k)$ by q_r replicas in view v. q_r is a parameter fixed for the protocol instance. We call $\mathcal{C}_v^{q_r}(B_k)$ a certificate for B_k from view v. Certified blocks are ranked first by the views in which they are certified and then by their heights. In other words, a block B_k certified in view v is ranked *higher* than a block $B_{k'}$ certified in view v' if either (i) $v > v'$ or (ii) $v = v'$ and $k > k'$.

Locked blocks. At any time, a replica locks the highest certified block to its knowledge. During the protocol execution, each replica keeps track of all signatures for all blocks and keeps updating its locked block. Looking ahead, the notion of locked block will be used to guard the safety of a client commit.

5.2 Replica Protocol

The replica protocol progresses in a view-by-view fashion. Each view has a designated leader who is responsible for driving consensus on a sequence of blocks. Leaders can be chosen statically, e.g., round robin, or randomly using more sophisticated techniques [8, 23]. In our description, we assume a round robin selection of leaders, i.e., (v mod n) is the leader of view v.

Let v be the current view number and replica L be the leader in this view. Perform the following steps in an iteration.

1. **Propose.** ▷ Executed by the leader of view v

   The leader L broadcasts $\langle \mathsf{propose}, B_k, v, \mathcal{C}_{v'}^{q_r}(B_{k-1}), \mathcal{S} \rangle_L$. Here, $B_k := (b_k, h_{k-1})$ is the newly proposed block and it should extend the highest certified block known to L. In the steady state, an honest leader L would extend the previous block it proposed, in which case $v' = v$ and $\mathcal{S} = \perp$. Immediately after a view change, L determines the highest certified block from the status $\mathcal{S}$ received during the view change.

2. **Vote.** ▷ Executed by all replicas

   When a replica R receives a valid proposal $\langle \mathsf{propose}, B_k, v, \mathcal{C}_{v'}^{q_r}(B_{k-1}), \mathcal{S} \rangle_L$ from the leader L, R broadcasts the proposal and a vote $\langle \mathsf{vote}, B_k, v \rangle_R$ if (i) the proposal is the first one in view v, and it extends the highest certified block in $\mathcal{S}$, or (ii) the proposal extends the last proposed block in the view.

   In addition, replica R records the following based on the messages it receives.

   - R keeps track of the number of votes received for this block in this view as $q_{B_k,v}$.
   - If block B_{k-1} has been proposed in view v, R marks B_{k-1} as a locked block and records the locked time as $\mathsf{t\text{-}lock}_{k-1,v}$.
   - If a block equivocating B_{k-1} is proposed by L in view v (possibly received through a vote), R records the time $\mathsf{t\text{-}equiv}_{k-1,v}$ at which the equivocating block is received.

   The replica then enters the next iteration. If the replica observes no progress or equivocating blocks in the same view v, it stops voting in view v and sends $\langle \mathsf{blame}, v \rangle_r$ message to all replicas.

Figure 2: Flexible BFT steady state protocol.

At a high level, the protocol does the following: The leader proposes a block to all replicas. The replicas vote on it if safe to do so. The block becomes certified once q_r replicas vote on it. The leader will then propose another block extending the previous one, chaining blocks one after another at increasing heights. Unlike regular consensus protocols where replicas determine when a block is committed, in Flexible BFT, replicas only certify blocks while committing is offloaded to the clients. If at any time replicas detect malicious leader behavior or lack of progress in a view, they blame the leader and engage in a view change protocol to replace the leader and move to the next view. The new leader collects a status from different replicas and continues to propose blocks based on this status. We explain the steady state and view change protocols in more detail below.

Steady state protocol. The steady state protocol is described in Figure 2. In the steady state, there is a unique leader who, in an iteration, proposes a block, waits for votes from q_r replicas and moves to the next iteration. In the steady state, an honest leader always extends the previous block it proposed. Immediately after a view change, since the previous leaders could have been Byzantine and may have proposed equivocating blocks, the new leader needs to determine a safe block to propose. It does so by collecting a status of locked blocks from q_r replicas denoted by $\mathcal{S}$ (described in the view change protocol).

For a replica R in the steady state, on receiving a proposal for block B_k, a replica votes for it if it extends the previous proposed block in the view or if it extends the highest certified block in $\mathcal{S}$. Replica R can potentially receive blocks out of order and thus receive B_k before its ancestor blocks. In this case, replica R waits until it receives the ancestor blocks, verifies the validity of those blocks and B_k before voting for B_k. In addition, replica R records the following to aid a client commit:

- **Number of votes.** It records the number of votes received for B_k in view v as $q_{B_k,v}$. Observe that votes are broadcast by all replicas and the number of votes for a block can be greater than q_r. $q_{B_k,v}$ will be updated each time the replica hears about a new vote in view v.

- **Lock time.** If B_{k-1} was proposed in the same view v, it locks B_{k-1} and records the locked time as $\mathsf{t\text{-}lock}_{k-1,v}$.
- **Equivocation time.** If the replica ever observes an equivocating block at height k in view v through a proposal or vote, it stores the time of equivocation as $\mathsf{t\text{-}equiv}_{k,v}$.

Looking ahead, the locked time $\mathsf{t\text{-}lock}_{k-1,v}$ and equivocation time $\mathsf{t\text{-}equiv}_{k-1,v}$ will be used by clients with synchrony assumptions to commit, and the number of votes $q_{B_k,v}$ will be used by clients with partial-synchrony assumptions to commit.

Leader monitoring. If a replica detects a lack of progress in view v or observes malicious leader behavior such as more than one height-k blocks in the same view, it blames the leader of view v by broadcasting a $\langle \mathsf{blame}, v \rangle$ message. It quits view v and stops voting and broadcasting blocks in view v. To determine lack of progress, the replicas may simply guess a time bound for message arrival or use increasing timeouts for each view [9].

View change. The view change protocol is described in Figure 3. If a replica gathers q_r $\langle \mathsf{blame}, v \rangle$ messages from distinct replicas, it forwards them to all other replicas and enters a new view $v+1$ (Step (i)). It records the time at which it received the blame certificate as $\mathsf{t\text{-}viewchange}_v$. Upon entering a new view, a replica reports to the leader of the new view L' its locked block and transitions to the steady state (Step (ii)). q_r status messages form the status $\mathcal{S}$. The first block L' proposes in the new view should extend the highest certified block among these q_r status messages.

Let L and L' be the leaders of views v and $v+1$, respectively.

(i) **New-view.** Upon gathering q_r $\langle \mathsf{blame}, v \rangle$ messages, broadcast them and enter view $v+1$. Record the time as $\mathsf{t\text{-}viewchange}_v$.

(ii) **Status.** Suppose B_j is the block locked by the replica. Send a status of its locked block to the leader L' using $\langle \mathsf{status}, v, B_j, \mathcal{C}_{v'}^{q_r}(B_j) \rangle$ and transition to the steady state. Here, v' is the view in which B_j was certified.

Figure 3: Flexible BFT view change protocol.

5.3 Client Commit Rules

(CR1) Partially-synchronous commit. A block B_k is committed under the partially synchronous rule with parameter q_c iff there exist $l \geq k$ and v such that

(a) $\mathcal{C}_v^{q_r}(B_l)$ and $\mathcal{C}_v^{q_r}(B_{l+1})$ exist where B_{l+1} extends B_l and B_k (if $l = k$, $B_l = B_k$).

(b) $q_{B_l,v} \geq q_c$ and $q_{B_{l+1},v} \geq q_c$.

(CR2) Synchronous commit. A block B_k is committed assuming $\Delta-$synchrony iff the following holds for q_r replicas. There exist $l \geq k$ and v (possibly different across replicas) such that,

(a) $\mathcal{C}_v^{q_r}(B_l)$ exists where B_l extends B_k (if $l = k$, $B_l = B_k$).

(b) An undisturbed-2Δ period is observed after B_{l+1} is obtained, i.e., no equivocating block or view change of view v were observed before 2Δ time after B_{l+1} was obtained, i.e.,

$$\min(\mathsf{current\text{-}time}, \mathsf{t\text{-}equiv}_{l,v}, \mathsf{t\text{-}viewchange}_v) - \mathsf{t\text{-}lock}_{l,v} \geq 2\Delta$$

Figure 4: Flexible BFT commit rules

As mentioned in the introduction, Flexible BFT supports clients with different assumptions. Clients in Flexible BFT learn the state of the protocol from the replicas and based on their own assumptions determine whether a block has been committed. Broadly, we supports two types of clients: those who believe in synchrony and those who believe in partial synchrony.

5.3.1 Clients with Partial-Synchrony Assumptions (CR1)

A client with partial-synchrony assumptions deduces whether a block has been committed by based on the number of votes received by a block. A block B_l (together with its ancestors) is committed with parameter q_c iff B_l and its immediate successor both receive $\geq q_c$ votes in the same view.

Safety of CR1. A CR1 commit based on q_c votes is safe against $< q_c + q_r - 1$ faulty replicas (Byzantine plus alive-but-corrupt). Observe that if B_l gets q_c votes in view v, due to flexible quorum intersection, a conflicting block cannot be certified in view v, unless $\geq q_c + q_r - 1$ replicas are faulty. Moreover, B_{l+1} extending B_l has also received q_c votes in view v. Thus, q_c replicas lock block B_l in view v. In subsequent views, honest replicas that have locked B_l will only vote for a block that equals or extends B_l unless they unlock. However, due to flexible quorum intersection, they will not unlock unless $\geq q_c + q_r - 1$ replicas are faulty. Proof of Lemma 1 formalizes this argument.

5.3.2 Client with Synchrony Assumptions (CR2)

Intuitively, a CR2 commit involves q_r replicas collectively stating that no "bad event" happens within "sufficient time" in a view. Here, a bad event refers to either leader equivocation or view change (the latter indicates sufficient replicas believe leader is faulty) and the "sufficient time" is 2Δ; where Δ is a synchrony bound chosen by the client. More formally, a replica states that a synchronous commit for block B_k for a given parameter Δ (set by a client) is satisfied iff the following holds. There exists B_{l+1} that extends B_l and B_k, and the replica observes an undisturbed-2Δ period after obtaining B_{l+1} during which (i) no equivocating block is observed, and (ii) no blame certificate/view change certificate for view v was obtained, i.e.,

$$\min(\mathsf{current\text{-}time}, \mathsf{t\text{-}equiv}_{l,v}, \mathsf{t\text{-}viewchange}_v) - \mathsf{t\text{-}lock}_{l,v} \geq 2\Delta$$

where $\mathsf{t\text{-}equiv}_{l,v}$ denotes the time equivocation for B_l in view v was observed (∞ if no equivocation), $\mathsf{t\text{-}viewchange}_v$ denotes the time at which view change happened from view v to $v+1$ (∞ if no view change has happened yet), and $\mathsf{t\text{-}lock}_{l,v}$ denotes the time at which B_l was locked (or B_{l+1} was proposed) in view v. Note that the client does not require the q_r fraction of replicas to report the same height l or view v.

Safety of CR2. A client believing in synchrony assumes that all messages between replicas arrive within Δ time after they were sent. If the client's chosen Δ is a correct upper bound on message delay, then a CR2 commit is safe against q_r faulty replicas (Byzantine plus alive-but-corrupt), as we explain below. If less than q_r replicas are faulty, at least one honest replica reported an *undisturbed*-2Δ period. Let us call this honest replica h and analyze the situation from h's perspective to explain why an undisturbed 2Δ period ensures safety. Observe that replicas in Flexible BFT forward the proposal when voting. If Δ-synchrony holds, every other honest replica learns about the proposal B_l at most Δ time after h learns about it. If any honest replica voted for a conflicting block or quit view v, h would have known within 2Δ time.

5.4 Safety and Liveness

We introduce the notion of *direct* and *indirect* commit to aid the proofs. We say a block is committed *directly* under **CR1** if the block and its immediate successor both get q_c votes in the same view. We say a block is committed *directly* under **CR2** if some honest replica reports an undisturbed-2Δ period after its successor block was obtained. We say a block is committed *indirectly* if neither condition applies to it but it is committed as a result of a block extending it being committed directly. We remark that the direct commit notion, especially for **CR2**, is merely a proof technique. A client cannot tell whether a replica is honest, and thus has no way of knowing whether a block is directly committed under **CR2**.

Lemma 1. *If a client directly commits a block B_l in view v using a correct commit rule, then a certified block that ranks no lower than $\mathcal{C}_v^{q_r}(B_l)$ either equals [Kartik: weird wording. can we say equivocation does not exist in this view and higher ranked blocks extend?] or extends B_l.*

Proof. To elaborate on the lemma, a certified block $\mathcal{C}_{v'}^{q_r}(B'_{l'})$ ranks no lower than $\mathcal{C}_v^{q_r}(B_l)$ if either (i) $v' = v$ and $l' \geq l$, or (ii) $v' > v$. We need to show that if B_l is directly committed, then any certified block that ranks no lower either equals or extends B_l. We consider the two commit rules separately. For both commit rules, we will use induction on v' to prove the lemma.

For **CR1** with parameter q_c to be correct, flexible quorum intersection needs to hold, i.e., the fraction of faulty replicas must be less than $q_c + q_r - 1$. B_l being directly committed under **CR1** with parameter q_c implies that there are q_c votes in view v for B_l and B_{l+1} where B_{l+1} extends B_l.

For the base case, a block $B'_{l'}$ with $l' \geq l$ that does not extend B_l cannot get certified in view v, because that would require $q_c + q_r - 1$ replicas to vote for two equivocating blocks in view v.

Next, we show the inductive step. Note that q_c replicas voted for B_{l+1} in view v, which contains $\mathcal{C}_v^{q_r}(B_l)$. Thus, they lock B_l or a block extending B_l by the end of view v. Due to the inductive hypothesis, any certified block that ranks equally or higher from view v up to view v' either equals or extends B_l. Thus, by the end of view v', those q_c replicas still lock B_l or a block extending B_l. Since the total fraction of faults is less than $q_c + q_r - 1$, the status $\mathcal{S}$ shown by the leader of view $v' + 1$ must include a certificate for B_l or a block extending it; moreover, any certificate that ranks equal to or higher than $\mathcal{C}_v^{q_r}(B_l)$ is for a block that equals or extends B_l. Thus, only a block that equals or extends B_l can gather votes from those q_c replicas in view $v' + 1$ and only a block that equals or extends B_l can get certified in view $v' + 1$.

For **CR2** with synchrony bound Δ to be correct, Δ must be an upper bound on worst case message delay and the fraction of faulty replicas is less than q_r. B_l being directly committed under **CR2** with Δ-synchrony implies that at least one honest replica voted for B_{l+1} extending B_l in view v, and did not hear an equivocating block or view change within 2Δ time after that. Call this replica h. Suppose h voted for B_{l+1} extending B_l in view v at time t, and did not hear an equivocating block or view change by time $t + 2\Delta$.

We first show the base case: a block $B'_{l'}$ with $l' \geq l$ certified in view v must equal or extend B_l. Observe that if $B'_{l'}$ with $l' \geq l$ does not equal or extend B_l, then it equivocates B_l. No honest replica voted for $B'_{l'}$ before time $t + \Delta$, because otherwise h would have received the vote for $B'_{l'}$ by time $t + 2\Delta$, No honest replica would vote for $B'_{l'}$ after time $t + \Delta$ either, because by then they would have received (from h) and voted for B_l. Thus, $B'_{l'}$ cannot get certified in view v.

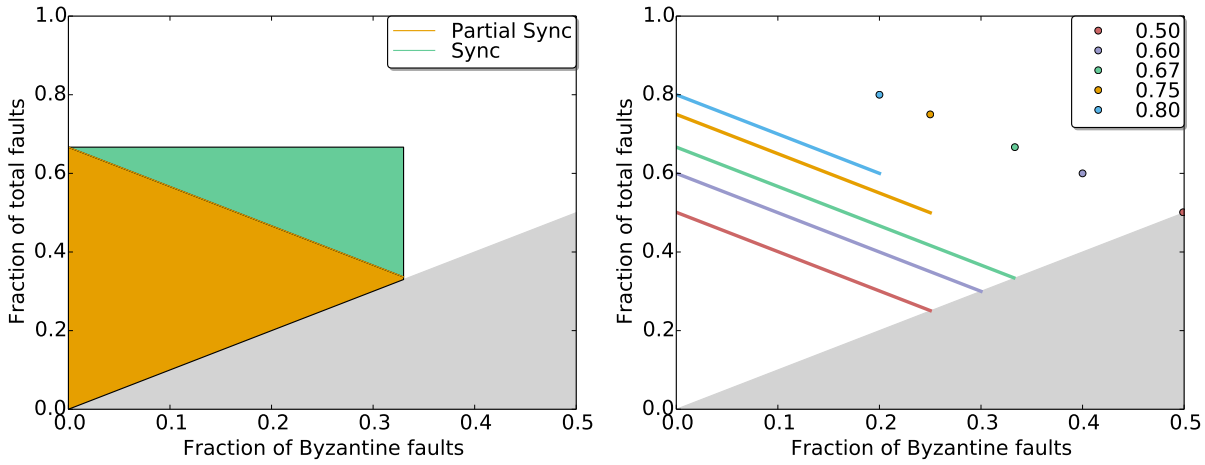
We then show the inductive step. Because h did not hear view change by time $t + 2\Delta$, all honest replicas are still in view v by time $t + \Delta$, which means they all receive B_{l+1} from h by the end of view v. Thus, they lock B_l or a block extending B_l by the end of view v. Due to the inductive hypothesis, any certified block that ranks equally or higher from view v up to view v' either equals or extends B_l. Thus, by the end of view v', all honest replicas still lock B_l or a block extending B_l. Since the total fraction of faults is less than q_r, the status $\mathcal{S}$ shown by the leader of view $v' + 1$ must include a certificate for B_l or a block extending it; moreover, any certificate that ranks equal to or higher than $\mathcal{C}_v^{q_r}(B_l)$ is for a block that equals or extends B_l. Thus, only a block that equals or extends B_l can gather honest votes in view $v' + 1$ and only a block that equals or extends B_l can get certified in view $v' + 1$. □

Theorem 2 (Safety)**.** *Two clients with correct commit rules commit the same block B_k for each height k.*

Proof. Suppose for contradiction that two distinct blocks B_k and B'_k are committed at height k. Suppose B_k is committed as a result of B_l being directly committed in view v and B'_k is committed as a result of $B'_{l'}$ being directly committed in view v'. This implies B_l is or extends B_k; similarly, $B'_{l'}$ is or extends B'_k. Without loss of generality, assume $v \leq v'$. If $v = v'$, further assume $l \leq l'$ without loss of generality. By Lemma 1, the certified block $\mathcal{C}_{v'}^{q_r}(B'_{l'})$ must equal or extend B_l. Thus, $B'_k = B_k$. □

Theorem 3 (Liveness)**.** *If all clients have correct commit rules, they all keep committing new blocks.*

Proof. By the definition of alive-but-corrupt faults, if they cannot violate safety, they will preserve liveness. Theorem 2 shows that if all clients have correct commit rules, then safety is guaranteed *even if alive-but-corrupt replicas behave arbitrarily.* Thus, once we proved safety, we can treat alive-but-corrupt replicas as honest when proving liveness.



(a) **Clients supported for $q_r = 2/3$.**

(b) **Clients supported by Flexible BFT at different q_r's.** The legend represents the different q_r values.

Figure 5

Observe that a correct commit rule tolerates at most $1 - q_r$ Byzantine faults. If a Byzantine leader prevents liveness, there will be q_r blame messages against it, and a view change will ensue to replace the leader. Eventually, a non-Byzantine (honest or alive-but-corrupt) replica becomes the leader and drives consensus in new heights. If replicas use increasing timeouts, eventually, all non-Byzantine replicas stay in the same view for sufficiently long. When both conditions occur, if a client's commit rule is correct (either **CR1** and **CR2**), due to quorum availability, it will receive enough votes in the same view to commit. □

6 Discussion

As we have seen, three parameters q_r, q_c, and Δ determine the protocol. q_r is the only parameter for the replicas and is picked by the service administrator. The choice of q_r determines a set of client assumptions that can be supported. q_c and Δ are chosen by clients to commit blocks. In this section, we first discuss the client assumptions supported by a given q_r and then discuss the trade-offs between different choices of q_r.

6.1 Client Assumptions Supported by a Given q_r

Figure 5a represents the clients supported at $q_r = 2/3$. The x-axis represents Byzantine faults and the y-axis represents total faults (Byzantine plus alive-but-corrupt). Each point on this graph represents a client fault assumption as a pair: (Byzantine faults, total faults). The shaded gray area indicates an "invalid area" since we cannot have fewer total faults than Byzantine replicas. A missing dimension in this figure is the choice of Δ. Thus, the synchrony guarantee shown in this figure is for clients that choose a correct synchrony bound.

Clients with partial-synchrony assumptions can get fault tolerance on (or below) the starred orange line. The right most point on the line is $(1/3, 1/3)$, i.e., we tolerate less than a third of Byzantine replicas and no additional alive-but-corrupt replicas. This is the setting of existing partially synchronous consensus protocols [10, 9, 27]. Flexible BFT generalizes these protocols by giving clients the option of moving up-left along the line, i.e., tolerating fewer Byzantine and more total faults. By choosing $q_c > q_r$, a client tolerates $< q_c + q_r - 1$ total faults for safety and $\leq 1 - q_c$ Byzantine faults for liveness. In other words, as a client moves left, for every additional vote it requires, it tolerates one fewer Byzantine fault and one more total fault (i.e., two more alive-but-corrupt faults). The left most point on this line $(0, 2/3)$ tolerates no Byzantine replicas and the highest fraction of alive-but-corrupt replicas.

Moreover, for clients who believe in synchrony, if their Δ assumption is correct, they enjoy $1/3$ Byzantine tolerance and $2/3$ total tolerance represented by the green diamond. This is because synchronous commit rules are not parameterized by the number of votes received.

How do clients pick their commit rules? In Figure 5a, the shaded starred orange portion of the plot represent fault tolerance provided by the partially synchronous commit rule (CR1). Specifically, setting q_c to the total fault fraction yields the necessary commit rule. On the other hand, if a client's required fault tolerance lies in the circled green portion of the plot, then the synchronous commit rule (CR2) with an appropriate Δ picked by the client yields the necessary commit rule. Finally, if a client's target fault tolerance corresponds to the white region of the plot, then it is not achievable with this q_r.

Clients with incorrect assumptions and recovery. If a client has incorrect assumption with respect to the fault threshold or synchrony parameter Δ, then it can lose safety or liveness. If a client believing in synchrony picks too small a Δ and commits a value b, it is possible that a conflicting value b' may also be certified. Replicas may choose to extend the branch containing b', effectively reverting b and causing a safety violation. Whenever a client detects such a safety violation, it may need to revert some of its commits and increase Δ to recover.

For a client with partial-synchrony assumption, if it loses safety, it can update its fault model to move left along the orange starred line, i.e., tolerate higher total faults but fewer Byzantine. On the other hand, if it observes no progress as its threshold q_c is not met, then it moves towards the right. However, if the true fault model is in the circled green region in Figure 5a, then the client cannot find a partially synchronous commit rule that is both safe and live and eventually has to switch to using a synchronous commit rule.

Recall that the goal of alive-but-corrupt replicas is to attack safety. Thus, clients with incorrect assumptions may be exploited by alive-but-corrupt replicas for their own gain (e.g., by double-spending). When a client updates to a correct assumption and recovers from unsafe commits, their subsequent commits would be safe and final. This is remotely analogous to Bitcoin – if a client commits to a transaction when it is a few blocks deep and a powerful adversary succeeds in creating an alternative longer fork, the commit is reverted.

6.2 Comparing Different q_r Choices

We now look at the service administrator's choice at picking q_r. In general, the service administrator's goal is to tolerate a large number of Byzantine and alive-but-corrupt faults, i.e., move towards top and/or right of the figure. Figure 5b shows the trade-offs in terms of clients supported by different q_r values in Flexible BFT.

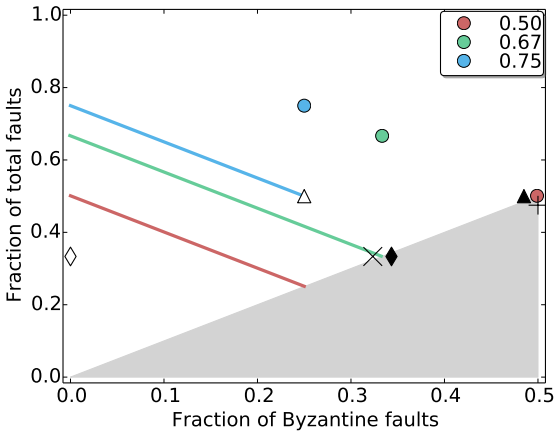
First, it can be observed that for clients with partial-synchrony assumptions, $q_r \geq 2/3$ dominates $q_r < 2/3$. Observe that the fraction of Byzantine replicas (B) are bounded by $B < q_c + q_r - 1$ and $B \leq 1 - q_c$, so $B \leq q_r/2$. Thus, as q_r decreases, Byzantine fault tolerance decreases. Moreover, since the total fault tolerance is $q_c + q_r - 1$, a lower q_r also tolerates a smaller fraction of total faults for a fixed q_c.

For $q_r \geq 2/3$ or for clients believing in synchrony, no value of q_r is Pareto optimal. For clients with partial-synchrony assumptions, as q_r increases, the total fault tolerance for safety increases. But since $q_c \geq q_r$, we have $B \leq 1 - q_r$, and hence the Byzantine tolerance for liveness decreases. For clients believing in synchrony, the total fault tolerance for safety is $< q_r$ and the Byzantine fault tolerance for liveness is $\geq 1 - q_r$. In both cases, the choice of q_r represents a safety-liveness trade-off.

7 Related Work

Flexible BFT and existing consensus protocols. Figure 6 shows a comparison of Flexible BFT with some of the existing consensus protocols. The two axes represent Byzantine replicas and total adversarial replicas that can be tolerated. The three different colors (red, green, blue) represent three possible instantiations of Flexible BFT at different q_r's. The lines represent the safety threshold for partially synchronous clients whereas the colored circles represent the same for synchronous clients. The locus of points on a given color represents all client assumptions supported for that q_r, representing the heterogeneity of clients supported. The different uncolored shapes, $+, \times, \triangle, \blacktriangle, \Diamond, \blacklozenge$, represent the clients supported by existing consensus protocols.

BFT protocols have been studied in various settings, including synchrony [25, 13, 4, 1], partial synchrony [9, 26, 21, 16, 27, 6], and asynchrony [24]. These works all assume a homogeneous world where



+ Partial Synchrony [9, 26, 21, 16, 27, 6]
× Synchrony [25, 13, 4, 1]
▲ Thunderella, Sync HotStuff (△: optimistic) [25, 4]
♦ Zyzzyva, SBFT (◇: optimistic) [16]

Figure 6: **Comparing Flexible BFT to existing consensus protocols.** The legend represent different q_r values.

all parties involved (replicas and clients) have the same set of assumptions. In Figure 6, the partially synchronous protocols [9, 27, 6] that tolerate one-third Byzantine faults can all be represented by the '+' symbol at $(1/3, 1/3)$. Similarly, synchronous protocols [13, 3, 1] that tolerate one-half Byzantine faults are represented by the '×' symbol at $(1/2, 1/2)$. It is worth noting that some of these works employ two commit rules that differ in number of votes or synchrony [21, 16, 25, 4]. For instance, Thunderella and Sync HotStuff optimistically commit in an asynchronous fashion based on quorums of size $\geq 3/4$, as represented by a hollow triangle at $(1/4, 1/2)$. Similarly, FaB [21], Zyzzyva [16] and SBFT [12] optimistically commit when they receive all votes but wait for two rounds of votes otherwise. Despite the presence of two commit rules, these protocols are in the homogeneous model since all parties involved (replicas and clients) respect both commit rules and make the same set of assumptions.

Consensus protocols with heterogeneity. A simple and different notion of heterogeneity exists in Bitcoin's probabilistic commit rule. For example, one recipient may consider a transaction committed after six confirmations while another may require only one confirmation. The notion of heterogeneity has also been discussed informally at public blockchain forums. Another example of heterogeneity is considered in XFT[18] where clients can assume either crash faults under partial synchrony or Byzantine faults under synchrony. Yet another notion of heterogeneity is considered by the federated Byzantine consensus model and the Stellar protocol [22]. The Stellar protocol allows nodes to pick their own quorums. Our Flexible BFT approach instead considers heterogeneous clients in terms of alive-but-corrupt adversaries and synchrony. The model and techniques in [22] and our paper are largely orthogonal and complementary.

Flexible BFT vs Flexible Paxos. Flexible Paxos by Howard et al. [14] observes that Paxos may use non-intersecting quorums within a view but an intersection is required across views. Our Flexible Quorum Intersection (b) can be viewed as its counterpart in the Byzantine and alive-but-corrupt setting. In addition, Flexible BFT applies the flexible quorum idea to support heterogeneous clients with different fault model and timing assumptions.

Rational fault model. Our alive-but-corrupt adversary is similar to a rational adversary in the BAR model [5], though they assume no collusion among rational replicas. Game theoretical modeling and analysis with collusion have been performed to other problems such as secret sharing and multiparty computation [2, 19, 11, 15].

8 Conclusion and Future Work

We present Flexible BFT, a protocol that supports heterogeneous clients with different assumptions to co-exist and use the same ledger. Flexible BFT allows the clients to tolerate combined (Byzantine plus alive-but-corrupt) faults exceeding 1/2 and 1/3 for synchrony and partial synchrony respectively. At a technical level, under synchrony, we show a synchronous protocol where the replicas execute a network speed protocol and only the commit rule uses the synchrony assumption. For partial synchrony, we introduce the notion of Flexible Byzantine Quorums by deconstructing existing BFT protocols to understand the role played by the different quorums. We combine the two to form Flexible BFT which obtains the best of both worlds.

Our liveness proof in Section 5.4 employs a strong assumption that all clients have correct commit rules. This is because our alive-but-corrupt fault model did not specify what these replicas would do if they can violate safety for some clients. In particular, they may stop helping liveness. However, we believe this will not be a concern once we move to a more realistic rational model. In that case, the best strategy for alive-but-corrupt replicas is to attack the safety of clients with unsafe commit rules while preserving liveness for clients with correct commit rules. Such an analysis in the rational fault model remains interesting future work. Our protocol also assumes that all replicas have clocks that advance at the same rate. It is interesting to explore whether our protocol can be modified to work with clock drifts.

Acknowledgement

We thank Ittai Abraham and Ben Maurer for many useful discussions on Flexible BFT.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience. In *Financial Cryptography and Data Security (FC)*, 2019.

[2] Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 53–62. ACM, 2006.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Dfinity consensus, explored. Cryptology ePrint Archive, Report 2018/1153, 2018.

[4] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Synchronous smr with 2δ latency and optimistic responsiveness. Cryptology ePrint Archive, Report 2019/270, 2019. `https://eprint.iacr.org/2019/270`.

[5] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *ACM SIGOPS operating systems review*, volume 39, pages 45–58. ACM, 2005.

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

[7] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

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

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

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

[11] S Dov Gordon and Jonathan Katz. Rational secret sharing, revisited. In *International Conference on Security and Cryptography for Networks*, pages 229–241. Springer, 2006.

[12] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. *arXiv preprint arXiv:1804.01626*, 2018.

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

[14] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In *OPODIS*, volume 70 of *LIPIcs*, pages 25:1–25:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[15] Gillat Kol and Moni Naor. Cryptography and game theory: Designing protocols for exchanging information. In *Theory of Cryptography Conference*, pages 320–339. Springer, 2008.

[16] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

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

[18] Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 485–500. USENIX Association, 2016.

[19] Anna Lysyanskaya and Nikos Triandopoulos. Rationality and adversarial behavior in multi-party computation. In *Annual International Cryptology Conference*, pages 180–197. Springer, 2006.

[20] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 569–578, New York, NY, USA, 1997. ACM.

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

[22] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus, 2015.

[23] Silvio Micali. Algorand: The efficient and democratic ledger. arXiv:1607.01341, 2016.

[24] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

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

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

[27] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. *arXiv preprint arXiv:1803.05069*, 2018.