The Istanbul BFT Consensus Algorithm

Henrique Moniz
Quorum Engineering

May 12, 2020

1 Introduction

In this paper, we present *Istanbul BFT* (IBFT), a Byzantine fault-tolerant (BFT) consensus algorithm that is used for implementing state-machine replication in the *Quorum* blockchain. Quorum is an open source permissioned blockchain platform. It is based on Ethereum and designed for enterprise applications.

IBFT was initially proposed informally in EIP-650 [20]. That first proposal had safety issues where two correct processes could decide different values. A revision addressed these safety issues but introduced liveness issues where some executions could lead to a deadlock [10, 23]. This paper offers a precise and *correct* description of the algorithm along with correctness proofs, solving both the liveness and safety issues of earlier versions. Saltini et al. provide an alternative solution to the correctness issues in the initial IBFT proposal by applying the PBFT protocol [7] to a blockchain application [24].

IBFT belongs to a class of BFT algorithms that assume a partially synchronous communication model [13]. Under this model messages can be arbitrarily delayed, but the system is characterized by also having periods of good communication in which messages are timely delivered. The IBFT algorithm is deterministic, leader-based, and optimally resilient - tolerating f faulty processes out of n, where $n \geq 3f + 1$ [21]. During periods of good communication, IBFT achieves termination in three message delays and has a communication complexity of $O(n^2)$.

IBFT follows from the line of work started with PBFT [7], which was the first practical algorithm designed for this model. More recently, a number of

	Communication Complexity		**Latency**
	Normal Case	View Change	Message Delays
DLS [13]	$O(n^4)$	$O(n^4)$	$O(n)$
PBFT [7]	$O(n^2)$	$O(n^3)$	3
Zyzzyva [16]	$O(n)$	$O(n^3)$	1 / 3*
Spinning [27]	$O(n^2)$	$O(n^3)$	3
SBFT [15]	$O(cn)^\dagger$	$O(n^2)$	5 / 7
HotStuff [28]	$O(n)$	$O(n)$	8
IBFT (Sec. 4)	$O(n^2)$	$O(n^2)$	3

Table 1: Performance of related algorithms when communication is timely. Message delays for dual-mode protocols are shown as x / y where x is for the optimistic environment and y otherwise.
*Zyzzyva requires, for the slow path, waiting for the maximum network delay Δ in addition to the 3 message delays. Additionally, its fast path has known correctness issues [1].
$^\dagger O(fn)$ if applying the recommended heuristic. See Section 2 for context.

consensus algorithms have been proposed under the same class with the aim of being used specifically in blockchain systems [4, 5, 15, 28].

Table 1 shows the communication complexity and latency to reach a decision for different algorithms under this class. The communication complexity is the asymptotic upper bound on the total number of bits exchanged by the algorithm. The latency is the number of message delays incurred by a correct process until termination. Both metrics pertain for a period of synchrony in which messages are timely delivered.

IBFT minimizes worst-case latency (i.e., the number of messages delays to termination without optimistic assumptions about the environment). It achieves termination in three message delays - matching PBFT, Zyzzyva, and Spinning - while improving on the communication complexity of these algorithms by exchanging a quadratic number of messages. HotStuff achieves linear communication complexity but it does so at a high cost in latency.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 defines the system model. Section 4 describes the IBFT algorithm in detail. Section 5 proves the correctness of IBFT. Finally, Section 6 concludes the paper.

2 Related Work

The problems of consensus and state machine replication (SMR) are closely associated with one another. Consensus requires processes in a distributed system to reach agreement on some value [26]. SMR requires agreement on a total order of commands [17, 25]. When consensus is solvable, so is SMR. As such, consensus is often used as a building block to implement SMR.

The problem of a distributed system reaching agreement in the presence of Byzantine process failures was first devised by Pease et al. [19, 21]. They also propose solutions for synchronous systems, where there is a known bound on the message transmission delays and the relative speeds of processes.

Of more practical utility are solutions to the problem of consensus in asynchronous systems, where there are no timing assumptions. Fischer et al., however, proved there there is no deterministic solution to consensus in asynchronous system where a single process is allowed to fail [14]. There is an abundant body of research dedicated to circumventing this impossibility result using different techniques. The most notable examples being partial synchrony [12, 13], failure detectors [2, 8], and randomization [3, 9, 22].

In this paper, we are concerned with implementing Byzantine fault-tolerant state machine replication in a partially synchronous system. We thus restrict our comparison to protocols that, like IBFT, assume a partially synchronous model, tolerate Byzantine process failures, and are optimally resilient (i.e., $n \geq 3f + 1$).

The partially synchronous model was introduced by Dwork et al. [13]. Along with it, they also proposed a Byzantine fault-tolerant consensus algorithm (DLS), which, although inefficient, proved that the problem has a solution in partially synchronous systems.

The PBFT algorithm was the first to provide a correct solution for state machine replication with Byzantine faults in a partially synchronous system where safety does not depend on timing assumptions [7]. This work was seminal in that it inspired a long line of algorithms that explore the design space within the same partially synchronous model.

The Zyzzyva algorithm introduced the idea of using speculative execution to improve performance [16]. Replicas optimistically adopt the order proposed by the primary and delegate to the clients the detection of inconsistencies, which help replicas resolve their state to one that is consistent with a total ordering of requests. Clement et al. later showed that this approach suffers from significant performance problems if even a single malicious client is present in the system [11]. As a solution, they propose a new algorithm - Aardvark - that makes more robust design decisions at a cost of best-case performance. More recently, Abraham et al. also demonstrated safety violations in the optimistic execution of Zyzzyva [1].

Spinning was the first algorithm to use a rotating leader replica [27], a concept later applied by many blockchain-motivated BFT algorithms. The leader replica is changed after every request execution instead of only when it is suspected to have failed.

More recently, we have seen a new wave of protocols that are motivated by their application to blockchain systems. We highlight Tendermint, SBFT, and HotStuff.

Tendermint is a protocol whose main novelty is that it does not have a separate round (i.e., view) change algorithm [4]. Replicas change to a new round $r + 1$ as part of the normal operation by reaching a decision on round r. Similarly to Spinning, this allows for leader rotation as part of the normal operation and not just when a leader is suspected to be faulty. The main drawback of Tendermint is that even if a round has timely communication and an honest leader, it does not guarantee that a decision will be reached within that round. This is because if a correct process is locked on a block b that is not the one being proposed, then the algorithm needs to keep advancing the view until it reaches one where the leader proposes b. This results in a total communication complexity of $O(n^3)$ and latency of $O(n)$.

SBFT is a dual-mode protocol, employing a faster optimistic protocol - inspired by Zyzzyva - when there are no faulty replicas and the system is synchronous, and a slower fallback protocol - similar to PBFT - otherwise [15]. SBFT makes use of the concept of a *collector*. During a communication round, each replica, instead of broadcasting its message, sends it to a designated replica that aggregates the messages from all replicas into a single message and broadcasts it. Messages are signed using threshold signatures, which allow for the aggregated message to have a constant size. Since a single slow or failed collector would be sufficient to make the system switch to the fallback, slower protocol, SBFT allows the optimistic protocol to tolerate a parameterized number c of slow or failed replicas out of $n = 3f + 2c + 1$. Thus, for any $c > 0$, the algorithm fails to achieve optimal resiliency. SBFT has $O(cn)$ communication complexity during the normal case and $O(n^2)$ complexity during view changes. If c is a constant, then this results in linear complexity during the normal case. It is unlikely, however, for c to remain a constant value as a system scales. The authors recommend $c \leq f/8$ as a good heuristic, in which case the communication complexity would be $O(fn)$.

HotStuff is another protocol that employs the concept of a collector combined with threshold signatures to reduce communication complexity [28]. Unlike SBT, however, it only uses the primary replica as the collector. Like Tendermint, HotStuff does not employ a separate view change protocol. Instead, the view is advanced as part of the normal execution. This allows it to achieve $O(n)$ communication. The trade-off is a higher number of message delays to reach a decision.

3 System Model

The system is composed by a known set of n processes $\Pi = \{p_1, p_2, ..., p_n\}$. A process that follows the algorithm is said to be *correct*. Otherwise, it is said to be *faulty*. A faulty process can behave in an arbitrary (i.e., Byzantine) way, including sending purposely wrong messages with the intent to obviate the correct execution of the algorithm. The number of faulty processes is constrained to f, such that $n \geq 3f + 1$

We assume a partially synchronous system, where there is an unknown upper bound Δ on execution and communication delays that holds after an unknown period of time called *global stabilization time* (or GST, for short) [13].

Processes communicate by sending messages over a network. Any message sent before GST can be arbitrarily delayed or lost by the network. Reaching GST, we assume the following: a message sent by a correct process at some time t, such that $t \geq GST$, is guaranteed to be delivered by all correct processes by time $t + \Delta$.

4 The IBFT Algorithm

We present the IBFT pseudocode in algorithms 1 to 4. The pseudocode depicts how a consensus instance identified by λ is executed by a correct process p_i. Algorithm 1 has the constants, state variables, and the START procedure, which starts a consensus instance λ on p_i. Algorithm 2 describes the normal case operation of IBFT, which happens during periods where the leader is correct and messages are timely delivered. Algorithm 3 details how round changes are performed, for when a leader is suspected to be faulty. Finally, Algorithm 4 has the predicates used for message justification, which guarantees correctness during round changes by ensuring that only safe values are proposed.

Algorithm 1 IBFT pseudocode for process p_i: constants, state variables, and ancillary procedures

```
1:  constants:
2:      $p_i$                                        ▷ The identifier of the process

3:  state variables:
4:      $\lambda_i$                                  ▷ The identifier of the consensus instance
5:      $r_i$                                        ▷ The current round
6:      $pr_i$                                       ▷ The round at which the process has prepared
7:      $pv_i$                                       ▷ The value for which the process has prepared
8:      $inputValue_i$                               ▷ The value passed as input to this instance

9:  timer:
10:     $timer_i$

11: procedure Start($\lambda$, $value$)
12:     $\lambda_i \leftarrow \lambda$
13:     $r_i \leftarrow 1$
14:     $pr_i \leftarrow \perp$
15:     $pv_i \leftarrow \perp$
16:     $inputValue_i \leftarrow value$
17:     if leader($h_i$, $r_i$) $= p_i$ then
18:         broadcast ⟨PRE-PREPARE, $\lambda_i$, $r_i$, $inputValue_i$⟩ message
19:     set $timer_i$ to running and expire after $t(r_i)$
```

4.1 Preliminaries

The IBFT algorithm solves the consensus problem, where all correct processes need to decide on some common value. More formally, each instance of IBFT guarantees the following properties:

Agreement. If a correct process decides some value v, then no correct process decides a value v' such that $v' \neq v$.

Validity. Given an externally provided predicate β, if a correct process decides some value v, then $\beta(v)$ is true.

Termination. Every correct process eventually decides.

Algorithm 2 IBFT pseudocode for process p_i: normal case operation

```
 1: upon receiving a valid ⟨PRE-PREPARE, $\lambda_i$, $r_i$, $value$⟩ message $m$ from LEADER($\lambda_i$,
    $round$) such that JustifyPrePrepare($m$) do
 2:     set $timer_i$ to running and expire after $t(r_i)$
 3:     broadcast ⟨PREPARE, $\lambda_i$, $r_i$, $value$⟩

 4: upon receiving a quorum of valid ⟨PREPARE, $\lambda_i$, $r_i$, $value$⟩ messages do
 5:     $pr_i \leftarrow r_i$
 6:     $pv_i \leftarrow value$
 7:     broadcast ⟨COMMIT, $\lambda_i$, $r_i$, $value$⟩

 8: upon receiving a quorum $Q_{commit}$ of valid ⟨COMMIT, $\lambda_i$, $round$, $value$⟩ messages do
 9:     set $timer_i$ to stopped
10:     Decide($\lambda_i, value, Q_{commit}$)
```

Algorithm 3 IBFT pseudocode for process p_i: round changes

```
 1: upon $timer_i$ is expired do
 2:     $r_i \leftarrow r_i + 1$
 3:     set $timer_i$ to running and expire after $t(r_i)$
 4:     broadcast ⟨ROUND-CHANGE, $\lambda_i$, $r_i$, $pr_i$, $pv_i$⟩

 5: upon receiving a set $F_{rc}$ of $f + 1$ valid ⟨ROUND-CHANGE, $\lambda_i$, $r_j$, _, _⟩ messages such
    that $\forall$⟨ROUND-CHANGE, $\lambda_i$, $r_j$, _, _⟩ $\in F_{rc} : r_j > r_i$ do
 6:     let ⟨ROUND-CHANGE, $h_i$, $r_{min}$, _, _⟩ $\in F_{rc}$ such that:
 7:             $\forall$⟨ROUND-CHANGE, $\lambda_i$, $r_j$, _, _⟩ $\in F_{rc} : r_{min} \leq r_j$
 8:     $r_i \leftarrow r_{min}$
 9:     set $timer_i$ to running and expire after $t(r_i)$
10:     broadcast ⟨ROUND-CHANGE, $\lambda_i$, $r_i$, $pr_i$, $pv_i$⟩

11: upon receiving a quorum $Q_{rc}$ of valid ⟨ROUND-CHANGE, $\lambda_i$, $r_i$, _, _⟩ messages such
    that LEADER($\lambda_i$, $r_i$) $= p_i \wedge$ JustifyRoundChange($Q_{rc}$) do
12:     if HighestPrepared($Q_{rc}$) $\neq \perp$ then
13:         let $v$ such that $(\_, v) =$ HighestPrepared($Q_{rc}$))
14:     else
15:         let $v$ such that $v = inputValue_i$
16:     broadcast ⟨PRE-PREPARE, $\lambda_i$, $r_i$, $v$⟩

    ▷ We can omit this if we assume some mechanism external to the consensus algorithm that ensures
    synchronization of decided values.
17: upon receiving a valid ⟨ROUND-CHANGE, $\lambda_i$, _, _, _⟩ message from $p_j \wedge p_i$ has decided
    by calling Decide($\lambda_i$,_ ,$Q_{commit}$) do
18:     send $Q_{commit}$ to process $p_j$
```

Algorithm 4 IBFT pseudocode for process p_i: message justification

```
1: predicate JustifyRoundChange($Q_{rc}$)
2:     return
           $\forall \langle$ROUND-CHANGE, $\lambda_i$, $r_i$, $pr_j$, $pv_j\rangle \in Q_{rc} : pr_j = \bot \wedge pv_j = \bot$
           $\vee$ received a quorum of valid $\langle$PREPARE, $\lambda_i$, $pr$, $pv\rangle$ messages such that:
               $(pr, pv) =$ HighestPrepared($Q_{rc}$)

3: predicate JustifyPrePrepare($\langle$PRE-PREPARE, $\lambda_i$, $round$, $value\rangle$)
4:     return
           $round = 1$
           $\vee$ received a quorum $Q_{rc}$ of valid $\langle$ROUND-CHANGE, $\lambda_i$, $round$, $pr_j$, $pv_j\rangle$ messages
             such that:
               $\forall \langle$ROUND-CHANGE, $\lambda_i$, $round$, $pr_j$, $pv_j\rangle \in Q_{rc} : pr_j = \bot \wedge pr_j = \bot$
               $\vee$ received a quorum of valid $\langle$PREPARE, $\lambda_i$, $pr$, $value\rangle$ messages such that:
                   $(pr, value) =$ HighestPrepared($Q_{rc}$)

   ▷ Helper function that returns a tuple $(pr, pv)$ where $pr$ and $pv$ are, respectively, the prepared round
   and the prepared value of the ROUND-CHANGE message in $Q_{rc}$ with the highest prepared round
5: function HighestPrepared($Q_{rc}$)
6:     return $(pr, pv)$ such that:
           $\exists \langle$ROUND-CHANGE, $\lambda_i$, $round$, $pr$, $pv\rangle \in Q_{rc}$ :
               $\forall \langle$ROUND-CHANGE, $\lambda_i$, $round$, $pr_j$, $pv_j\rangle \in Q_{rc} : pr_j = \bot \vee pr \geq pr_j$
```

Our validation condition deserves further explanation. It uses the notion of *external validity*, originally proposed by Cachin et al. [6]. The application calling the algorithm provides an arbitrary predicate β whose purpose is to ensure that the decided value is acceptable within the context of the application. For instance, a blockchain implementation might want to ensure that the decided value is a block containing legitimate transactions.

Each instance λ of the algorithm proceeds in rounds. During each round, one of the processes acts has a leader that tries to drive the execution to a common decision by proposing a value. During a *good* round, where communication is timely and the leader is not faulty (i.e., after GST), the algorithm guarantees that all correct processes will reach a decision. There is a function LEADER(λ, *round*) that identifies the leader. This function can be any deterministic mapping from λ and *round* to the identifier of a process as long as it allows $f + 1$ processes to eventually assume the leader role.

Messages. Messages are represented as tuples enclosed in angle brackets. There are four types of messages: `PRE-PREPARE`, `PREPARE`, `COMMIT`, and `ROUND-CHANGE`. The first three types are of the form $\langle$`message-type`, λ, r, *value*$\rangle$ - where λ is the consensus instance, r is the round, and *value* is the proposal value - and comprise the normal case operation of the algorithm. A `ROUND-CHANGE` message is of the form $\langle$`ROUND-CHANGE`, λ, r, *pr*, *pv*$\rangle$ - where λ is the consensus instance, r is the round, *pr* is the prepared round, and *pv* is the prepared value - and is used to ensure progress when the current leader is suspected to have failed or the communication is not timely.

State. The algorithm state is composed of five variables: the consensus instance λ_i, the round r_i, the prepared round pr_i, and the prepared value pv_i, and the input value $inputValue_i$.

The variable λ_i identifies the instance of the consensus algorithm being executed. It is set upon a call to the START procedure and it never changes throughout the execution. When the algorithm is used to implement state machine replication, the instances are numbered in a total order that determines the execution of commands. In a blockchain system, λ_i can correspond to the block number.

The variable r_i identifies the round in which process p_i is currently on and it starts at 1.

The prepared round pr_i and prepared value pv_i variables are, respectively, the highest round and the corresponding value for which p_i has *prepared*. During the execution of a consensus instance λ, we say that a process p_i has prepared for a round r and a value v if it receives a *quorum of valid messages*[1] of the form $\langle$`PREPARE`, λ, *round*, *value*$\rangle$. These variables are initialized with a default value $\perp$, which means that p_i has not prepared yet. This mechanism is essential for the safety of the algorithm and we explain how it works in Section 4.4.

Finally, $inputValue_i$ is simply the value passed as input to process p_i, which is saved in this variable.

Timer. In addition to the state variables, each correct process p_i also maintains a timer represented by $timer_i$, which is used to trigger a round change when the algorithm does not sufficiently progress. The timer can be in one of two states: `running` or `expired`. When set to `running`, it is also set a time $t(r_i)$, which is an exponential function of the round number r_i, after which the state changes to `expired`.

Upon rules. The main logic of the algorithm is expressed as a set of event-driven *upon* rules that are triggered when some condition is met. These are found exclusively in Algorithms 2 and 3. While not explicit in the pseudocode, we impose the restriction that, within an instance of the algorithm, each upon rule is triggered at most once for any round r_i. The only exception is the last rule on Algorithm 3 (line 17), which can be triggered any number of times.

A condition necessary to trigger many upon rules is receiving a *quorum of valid messages* that match a certain pattern. We say that a process has received a quorum of valid messages if it has received *valid* messages from $\lfloor \frac{n+f}{2} \rfloor + 1$ different processes. For instance, the upon rule in line 3 of Algorithm 2 is triggered when process p_i receives $\lfloor \frac{n+f}{2} \rfloor + 1$ valid messages from different processes that have type `PREPARE`, match instance λ_i and round r_i, and have the same *value*.

Validation. A correct process only accepts a message if it considers it to be *valid*. To be valid, a message must carry some proof of integrity and authentication of its sender such as a digital signature. The external validity

[1]We explain in the validation paragraph below what this entails.

predicate β must also be true for the value carried by the message. Furthermore, a $\langle$`ROUND-CHANGE`, λ, r, pr, $pv\rangle$ message to be valid needs for the prepared round to be smaller than the round, i.e., $pr < r$.

4.2 Normal case operation.

We now explain how the algorithm works during normal case operation, i.e., in some round where communication is timely and the leader is correct.

For any correct process p_i, an execution of an instance λ of the algorithm begins with a call to the Start procedure (Algorithm 1, line 11), which takes as input parameters the instance identifier λ and an input *value*. The procedure then initializes the state variables and if p_i is the leader for the current round, it broadcasts a `PRE-PREPARE` message proposing $inputValue_i$.

The remainder of the normal case operation is expressed in Algorithm 2. Upon receiving a valid `PRE-PREPARE` message from the leader for instance λ_i and current round r_i that carries a *justified*[2] value, a process p_i restarts the timer and broadcasts a `PREPARE` message (lines 1-3).

Upon receiving a quorum of valid `PREPARE` messages for instance λ_i and current round r_i carrying the same *value*, a process p_i updates its prepared round pr_i and prepared value pv_i variables to match the value of the received messages (lines 4-7). We now say that p_i has *prepared* for round r_i and value *value*. It then broadcasts a `COMMIT` message carrying *value*.

Finally, upon receiving a quorum of valid `COMMIT` messages for instance λ_i with the same *round* and *value*, a process p_i decides by calling an externally provided Decide function (lines 8-10).

4.3 Round changes

The previous section explains how the algorithm works under good conditions. The algorithm, however, must be able to tolerate arbitrary periods where communication is untimely or the leader is faulty. Round changes are how the algorithm ensures liveness by allowing different processes to assume the role of a leader. The pseudocode for round changes is expressed in Algorithm 3.

A round change is primarily triggered by the timer, which is available in each process for each consensus instance and represented by $timer_i$. If

[2]This is explained in Section 4.4.

the algorithm has not made sufficient progress for a process p_i to decide during some round r_i, then the timer will eventually expire (Algorithm 3, line 1). When this happens, p_i advances to round $r_i + 1$ and broadcasts a `ROUND-CHANGE` message. This message carries the values of the prepared round pr_i and prepared value pv_i variables, which will be used by the new leader to select a value to propose in a `PRE-PREPARE` message for $r_i + 1$. We explain how this value is selected in Section 4.4, which discusses message justification.

The upon rule at line 5 is also used for liveness. It is not strictly required in the sense that the algorithm would still be live without it, but it helps ensuring that processes do not wait too long before advancing to a new round. Whenever a process p_i receives a set of $f + 1$ valid `ROUND-CHANGE` messages with any round number higher than its current round r_i, it advances to the smallest round within that set and broadcasts a `ROUND-CHANGE` message.

The upon rule at line 11 starts the normal operation of the new round by having the leader broadcast a `PRE-PREPARE` message. From this point on, the algorithm resumes as specified by Algorithm 2.

Finally, the upon rule at line 17 ensures that any process p_j can catch up to a decision already made by process p_i by having p_i send p_j a quorum of `COMMIT` messages for λ_i.

4.4 Message Justification: Safety Across Rounds

While round changes ensure liveness, we also need to ensure that the proposal value chosen by the leader of a new round is safe. For this, we rely on the following property:

- If some correct process could have decided a value v at some round $r < r'$, then v must be the value proposed in a `PRE-PREPARE` message for round r'.

To select a proposal value that guarantees this property, the leader process, upon receiving a quorum Q_{rc} of `ROUND-CHANGE` messages for round r' (Algorithm 3, line 11), follows the logic expressed in lines 12-15. If there is any message in Q_{rc} with prepared round and prepared value not equal to $\perp$, then the leader chooses v as the prepared value pv of the message with the highest prepared round pr among the messages in Q_{rc}. Otherwise, if all the messages in Q_{rc} have prepared round and prepared value equal to $\perp$, then the leader chooses the input value that was passed to the START function and saved in the $inputValue_i$ variable.

Choosing the proposal value in this way ensures safety across rounds. This is because if a correct process decides some value v during a round r, then v will be chosen as the proposal value for round $r + 1$. To see why, say that a correct process decides v during round r. Then, at least $f + 1$ correct processes must have prepared for v during round r'. This implies that any quorum of `ROUND-CHANGE` messages will have at least one message with prepared round and prepared value set to r and v, respectively, and v will be chosen as the proposal value if the processes follow the algorithm. Since v is the proposal value for round $r+1$, it is not possible for a correct process to decide a different value during round $r+1$. A form of inductive reasoning applies for any subsequent rounds.

While choosing the proposal value in this way ensures safety across rounds, it is possible for a leader that is faulty to deviate from algorithm and send a `PRE-PREPARE` message with any arbitrary value. Likewise, any faulty process can send a `ROUND-CHANGE` message with an incorrect prepared round and prepared value to try to deceive a correct leader into choosing an unsafe proposal.

As such, to demonstrate that they are correctly constructed, these messages need a *justification.* A justification is a set of other messages that prove that the message being justified is congruent with the algorithm. For example, a message $\langle$`ROUND-CHANGE`, λ, r, pr, $pv\rangle$ sent by a process p_i, which states that p_i prepared for round pr and value pv, is only congruent with the algorithm if it exists a quorum of $\langle$`PREPARE`, λ, pr, $pv\rangle$ messages. A justification is piggybacked on the message being justified, but it is not part of the message itself.

The predicates JUSTIFYPREPREPARE and JUSTIFYROUNDCHANGE in Algorithm 4 express the justification logic, which we further explain below.

ROUND-CHANGE justification. Messages of the type `ROUND-CHANGE` are justified together as a set. A correct process p_i considers a quorum Q_{rc} of `ROUND-CHANGE` messages to be justified if one of the following conditions is true:

J1 All of the messages in Q_{rc} have prepared round and prepared value equal to $\perp$.

J2 The justification has a quorum of valid $\langle$`PREPARE`, λ_i, pr, $pv\rangle$ messages such that $\langle$`ROUND-CHANGE`, λ_i, r, pr, $pv\rangle$ is the message with the highest prepared round different than $\perp$ in Q_{rc}.

PRE-PREPARE justification. A `PRE-PREPARE` message for round 1 does not need justification. For any round $r > 1$, a $\langle$`PRE-PREPARE`, λ, r, $value\rangle$ message is justified if its justification has a quorum Q_{rc} of `ROUND-CHANGE` messages such that conditions **J1** or **J2** are true, and if **J2** is true, then $value = pv$.

5 Correctness

In this section we prove the correctness of IBFT. It is organized in three subsections, one for each property of the algorithm as specified in Section 4.1 - agreement, validity, and termination. For improved readability, proofs that require longer or more complex arguments are written using the structured proof format proposed by Lamport [18].

5.1 Agreement

The proof of the agreement is structured as Lemmas 1 and 2 and Theorem 3. Lemma 1 is used as support to prove agreement during the same round, and Lemma 2 to prove agreement across different rounds. Theorem 3 uses both lemmas to conclude the reasoning.

Agreement during the same round.

Lemma 1 *If some correct process prepares for a value v and round r, then no correct process prepares for a value v' and round r such that $v' \neq v$.*

For this proof we assume that a correct process has prepared for value v at round r and we demonstrate that no correct process can prepare for a different value v' at the same round because there is no quorum of `PREPARE` messages for v'.

1. A correct process received $\lfloor \frac{n+f}{2} \rfloor + 1$ valid $\langle$`PREPARE`, λ, r, $v\rangle$ messages.

   Proof: This follows from the assumption. To prepare for value v and round r, a correct process must receive $\lfloor \frac{n+f}{2} \rfloor + 1$ valid $\langle$`PREPARE`, λ, r, $v\rangle$ messages.

2. $f+1$ correct processes broadcasted a $\langle$`PREPARE`, λ, r, $v\rangle$ message.

   PROOF: This follows from 1. If a correct process received $\lfloor \frac{n+f}{2} \rfloor + 1$ valid $\langle$`PREPARE`, λ, r, $v\rangle$ messages, then $f+1$ of those messages must be been broadcasted by correct processes.

3. For any value $v' \neq v$, at most $\lfloor \frac{n+f}{2} \rfloor$ valid $\langle$`PREPARE`, h, r, $v'\rangle$ messages were broadcasted.

   PROOF: We have established in 2 that $f+1$ correct processes broadcasted a $\langle$`PREPARE`, λ, r, $v\rangle$ message. It follows that at most $\lfloor \frac{n+f}{2} \rfloor$ processes could have broadcasted a $\langle$`PREPARE`, λ, r, $v'\rangle$ message.

4. Q.E.D.

   PROOF: This follows from 3 since to prepare for a value v' and round r, a correct process requires $\lfloor \frac{n+f}{2} \rfloor + 1$ valid $\langle$`PREPARE`, λ, r, $v'\rangle$ messages.

Agreement across rounds. Lemma 1 showed that no two correct processes can prepare for different values on the same round. To prove agreement across rounds we will rely on the 2, which states that that once $f+1$ correct processes prepare for the same round r and value v, then a `PRE-PREPARE` message for the subsequent round $r+1$ must propose v to be justified.

Lemma 2 *If $f+1$ correct processes prepare for a value v and round r, then, for any $\langle$`PRE-PREPARE`, λ, r', $v'\rangle$ message m such that $r' > r$ and $v' \neq v$,* JUSTIFYPREPREPARE(m) *must be false.*

We prove this lemma by induction on r'. In the base case we prove the statement for $r' = r+1$. In the induction step we prove for $r' > r+1$ and assume that JUSTFIFYPREPREPARE(m) is false for any r'' such that $r+1 \leq r'' < r'$.

For both cases, the argument is constructed by demonstrating that JUSTIFYPREPREPARE must be false for any $\langle$`PRE-PREPARE`, λ, r', $v'\rangle$ message m. According to the definition of the predicate, JUSTIFYPREPREPARE(m) is true if there is a quorum Q_{rc} of $\lfloor \frac{n+f}{2} \rfloor + 1$ valid $\langle$`ROUND-CHANGE`, λ, $r+1$, pr_j, $pv_j\rangle$ messages such that one of the following two conditions is true, which correspond to J1 and J2 from Section 4.4:

(a) $\forall \langle$`ROUND-CHANGE`, λ_i, $r+1$, pr_j, $pv_j\rangle \in Q_{rc} : pr_j = \perp \wedge pr_j = \perp$

(b) There are $\lfloor\frac{n+f}{2}\rfloor+1$ valid $\langle$`PREPARE`, λ, r', $v'\rangle$ messages such that $v' \neq v$ and $(r', v') = \textsc{HighestPrepared}(Q_{rc})$

We show, both for the base case and for the induction step, that both (a) and (b) are false.

1. Base Case: $r' = r + 1$

   1.1. Prove: Condition (a) is false.

      1.1.1. Any quorum Q_{rc} includes some message from a correct process that prepared for round r and value v.

      Proof: A quorum Q_{rc} by definition has $\lfloor\frac{n+f}{2}\rfloor + 1$ messages. Since, by assumption, $f + 1$ correct processes prepared for round r and value v, it follows that Q_{rc} must include some message from those $f + 1$ processes.

      1.1.2. Q.E.D.

      Proof: By 1.1.1, Q_{rc} has some message from a correct process that prepared for round r and value v. It follows that $\exists\langle$`ROUND-CHANGE`, $\lambda_i, r{+}1, pr_j, pv_j\rangle \in Q_{rc} : pr_j \neq \bot \wedge pr_j \neq \bot$.

   1.2. Prove: Condition (b) is false.

   For the proof argument, we assume that condition (b) is true and show that it leads to a contradiction.

      1.2.1. Any quorum Q_{rc} includes some message from a correct process that prepared for round r and value v.

      Proof: A quorum Q_{rc} by definition has $\lfloor\frac{n+f}{2}\rfloor + 1$ messages. Since, by assumption, $f + 1$ correct processes prepared for round r and value v, it follows that Q_{rc} must include some message from those $f + 1$ processes.

      1.2.2. $r' = r$

      Proof: Since we assume that condition 2 is true, r' must be the highest prepared round value pr_j in a `ROUND-CHANGE` message in Q_{rc}. By 1.2.1, some `ROUND-CHANGE` message in Q_{rc} has prepared round r, which is also the highest prepared round value possible for the message to be valid. It follows that $r' = r$.

1.2.3. $f + 1$ correct processes broadcasted a $\langle$`PREPARE`, λ, r, $v\rangle$ message.

PROOF: This follows from the assumption that $f + 1$ correct processes prepared for round r and value v.

1.2.4. For any value $v' \neq v$, at most $\lfloor \frac{n+f}{2} \rfloor$ valid $\langle$`PREPARE`, h, r', $v'\rangle$ messages were broadcasted.

PROOF: This follows from 1.2.2 and 1.2.3.

1.2.5. Q.E.D.

PROOF: The assumption in condition 2 that there are $\lfloor \frac{n+f}{2} \rfloor + 1$ valid $\langle$`PREPARE`, λ, r', $v'\rangle$ messages is in contradiction with assertion 1.2.4.

2. INDUCTION STEP: $r' > r + 1$

We now reason about the induction step. We prove that, for any $\langle$`PRE-PREPARE`, λ, r', $v'\rangle$ message m such that $r' > r+1$ and $v' \neq v$, JUSTIFYPREPREPARE(m) is false. For this, we use the induction assumption, which is, for any $\langle$`PRE-PREPARE`, λ, r'', $v'\rangle$ message m' such that $r + 1 \leq r'' < r'$ and $v' \neq v$, JUSTIFYPREPREPARE(m') is false.

2.1. PROVE: Condition (a) is false.

PROOF: The same argument as 1.1 applies here.

2.2. PROVE: Condition (b) is false.

2.3. If $(pr, v') = \text{HIGHESTPREPARED}(Q_{rc})$, then $pr \geq r$. Additionally, $pr < r'$.

PROOF: By assumption, $f + 1$ correct processes prepared for value v and round r. This means that any quorum Q_{rc} has some `ROUND-CHANGE` message with prepared round p_j equal or higher than r. As such, for pr to be the highest prepared round value of any message in Q_{rc}, we must have $pr \geq r$. The assertion $pr < r'$ follows from the definition of a valid `ROUND-CHANGE` message.

2.4. There is no quorum of $\langle$`PREPARE`, λ, r, $v'\rangle$ messages such that $v' \neq v$.

PROOF: This follows from Lemma 1.

2.5. There is no quorum of $\langle$`PREPARE`, λ, r'', $v'\rangle$ messages such that $r + 1 \leq r'' < r'$ and $v' \neq v$.

PROOF: A correct process only broadcasts a $\langle$`PREPARE`, λ, r'', $v'\rangle$ if it accepts a $\langle$`PRE-PREPARE`, λ, r'', $v'\rangle$ message. By assumption, there is no such `PRE-PREPARE` message. It follows that it is not possible have a quorum of $\langle$`PREPARE`, λ, r'', $v'\rangle$ messages

2.6. Q.E.D.

PROOF: In 2.3 we established that $r \leq pr < r'$, and in 2.4 and 2.5 we established that there is no quorum of `PREPARE` messages with value $v' \neq v$ for any round from r to $r' - 1$.

Agreement Property. We now use the previous two lemmas to prove the agreement property of consensus in Theorem 3.

Theorem 3 *(Agreement) If a correct process p_i decides some value v, then no correct process p_j decides a value v' such that $v' \neq v$.*

PROOF: Let us assume without loss of generality that some correct process p_i is the first to decide, and decides value v during some round r. From Lemma 1 we can deduce that no correct process can decide v' during r because no correct process prepares for v' during r.

For any round r' such that $r' > r$, it follows from Lemma 2 that no correct process can decide v' during r' because there is no $\langle$`PRE-PREPARE`, λ, r', $v'\rangle$ message m such that JUSTIFYPREPREPARE(m) is true. Therefore, no $\langle$`PRE-PREPARE`, λ, r', $v'\rangle$ message is accepted by any correct process and the algorithm does not make progress towards a decision on v'.

5.2 Validity

Theorem 4 *(Validity) Given an externally provided predicate β, if a correct process decides some value v, then $\beta(v)$ is true.*

PROOF: To decide, a correct process must receive a quorum of valid $\langle$`COMMIT`, λ, r, $v\rangle$ messages. A message is only considered valid if it carries a proposal value v such that $\beta(v)$ is true. As such, if a correct process decides v, then $\beta(v)$ must be true.

5.3 Termination

Theorem 5 *(Termination) Every correct process decides.*

For this proof, we assume, without loss of generality, that a correct process p_i has not decided yet and demonstrate that it eventually decides.

1. p_i must eventually reach some round r (i.e., it sets $r_i = r$ and broadcasts a ⟨`ROUND-CHANGE`, λ, r, $\_$, $\_$⟩ message) after GST with a correct leader p_L.

   PROOF: Since by assumption p_i does not decide, its round timer must keep expiring indefinitely until it reaches round r or it receives $f + 1$ `ROUND-CHANGE` messages where r is the highest round.

We now have two cases to consider. One where some correct process p_j has already decided some value v, and one where no correct process has decided yet.

2. CASE: Some correct process p_j has decided.

   2.1. p_j receives the ⟨`ROUND-CHANGE`, λ, r, $\_$, $\_$⟩ message broadcast by p_i and sends to p_i a quorum of valid COMMIT messages for the same value v.

   PROOF: By 1, since after GST all messages sent by correct processes are timely delivered.

   2.2. Q.E.D.

   PROOF: After GST, p_i must receive the quorum of valid COMMIT messages for value v sent p_j and decide.

3. CASE: No correct process has decided.

   3.1. Every correct process eventually reaches round r and broadcasts a valid ⟨`ROUND-CHANGE`, λ, r, $\_$, $\_$⟩ message.

   PROOF: The argument is the same as in step 1, but generalized for all correct processes.

   3.2. The correct leader p_L broadcasts a valid and justified ⟨`PRE-PREPARE`, λ, r, v⟩ message.

   PROOF: After GST, p_L must receive every ⟨`ROUND-CHANGE`, λ, r, $\_$,

$\_\rangle$ message broadcast by a correct process. Since those messages are piggybacked with the necessary `PREPARE` messages for justification, p_L is able to construct a valid and justified $\langle$`PRE-PREPARE`, λ, r, $v\rangle$ message.

3.3. Q.E.D.

PROOF: Since it is after GST, the algorithm will follow through with its normal case operation and p_i will decide v at the end of round r.

6 Conclusion

In this paper we proposed IBFT, a simple algorithm for the consensus problem in Byzantine-fault tolerant systems that is implemented by the Quorum blockchain. IBFT assumes a partially synchronous model. With timely communication, IBFT terminates within three message delays and has $O(n^2)$ communication complexity during both normal case operation and round changes. To ensure safety across round changes, IBFT relies on a justification mechanism of proposed values that is critical in achieving quadratic communication complexity.

Acknowledgements

We would like to thank the following people for reviewing and providing thoughtful feedback on earlier versions of this manuscript: Alysson Bessani and Rodrigo Rodrigues at the University of Lisbon, Jitendra Bhurat and Samer Falah at J.P. Morgan, and Roberto Saltini at ConsenSys.

References

[1] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J. Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[2] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[3] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[4] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[5] V. Buterin and V. Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO 2001: Proceedings of the 21st International Conference on Cryptology*, pages 524–541, 2001.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, Feb. 1999.

[8] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[9] B. Chor and C. Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, pages 443–497. JAI Press, 1989.

[10] M. Chuan. Istanbul BFT's design cannot successfully tolerate fail-stop failures. https://github.com/jpmorganchase/quorum/issues/305, 2018.

[11] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168, 2009.

[12] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

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

[15] G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 568–580, 2019.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58, 2007.

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

[18] L. Lamport. How to write a 21st century proof. In *Journal of Fixed Point Theory and Applications*, volume 11, pages 43–63, 2012.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[20] Y.-T. Lin. Istanbul Byzantine Fault Tolerance. https://github.com/ethereum/EIPs/issues/650, June 2017.

[21] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[22] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[23] R. Saltini and D. Hyland-Wood. Correctness analysis of IBFT. *CoRR*, abs/1901.07160, 2019.

[24] R. Saltini and D. Hyland-Wood. IBFT 2.0: A safe and live variation of the IBFT blockchain consensus protocol for eventually synchronous networks. *CoRR*, abs/1909.10194, 2019.

[25] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[26] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *Computer Journal*, 25(6):8–17, 1992.

[27] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[28] M. Yin, D. Malkhi, M. Reiter, G. Gueta, and I. Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.