A Permit-Based Optimistic Byzantine Ledger

A Preprint

Roland Schmid
ETH Zürich, Switzerland
roschmi@ethz.ch

Roger Wattenhofer
ETH Zürich, Switzerland
wattenhofer@ethz.ch

ABSTRACT

PermitBFT solves the byzantine consensus problem for \( n \) nodes tolerating up to \( f < \frac{n}{3} \) byzantine nodes in the partially synchronous networking model; it is the first BFT protocol to achieve an optimistic latency of just 2 message delays despite tolerating byzantine failures throughout the "fast track". The design of PermitBFT relies on two fundamental concepts. First, in PermitBFT the participating nodes do not wait for a distinguished leader to act and subsequently confirm its actions, but send permits to the next designated block creator proactively. Second, PermitBFT achieves a separation of the decision powers that are usually concentrated on a single leader node. A block creator controls which transactions to include in a new block, but not where to append the block in the block graph.

Keywords byzantine consensus · byzantine fault tolerance · permissioned ledger · state machine replication

1 Introduction

A distributed system consists of \( n \) nodes. The system is byzantine fault tolerant (BFT) if it can tolerate at most \( f < \frac{n}{3} \) arbitrarily malicious (byzantine) nodes. BFT protocols have been studied in great detail since many decades, both in theory and practice. Nowadays, BFT protocols are the key to building "permissioned blockchains", an area traditionally known as "state machine replication" [20, 24].

The interest in BFT systems has first been reignited by Castro and Liskov, when they presented their “Practical” BFT (PBFT) system [5]. After PBFT, a large number of other BFT systems emerged. These systems try to minimize the delay until transactions are committed. They do not optimize the worst case, since the worst case happens rarely in practice. Rather, they minimize the delay with a varying degree of optimistic assumptions (e.g., no message timeouts, leader not byzantine, no node byzantine, transactions already pre-ordered).

In this paper we present PermitBFT, a new BFT protocol to build a permissioned ledger. The PermitBFT protocol inverts the process of previous BFT protocols, achieving a better bound for transaction commit delay. In a nutshell, PermitBFT works as follows.

Protocol Overview. The system consists of \( n \) authenticated nodes, i.e. all messages are signed. Initially, every node stores the genesis block that contains an ordering of the participating nodes.

All nodes permit a designated block creator to append a new block at a specific position in the block graph. While the position of the new block is given, a creator may freely include any non-conflicting transactions in the new block. If the creator creates a block in a timely manner, the nodes will allow the next creator to append to that block. This can be seen as the nodes voting on the position where to append the next block, thus ensuring that the block graph maintains some desired structural properties.

In reality, however, there might be disagreement among the nodes on where to append a new block. There are essentially two reasons for this: either a creator did not produce the next block in a timely manner (from the perspective of sufficiently many nodes), or a byzantine creator created multiple blocks at the same position.

If this happens, the PermitBFT algorithm will unify the system by allowing blocks to be created with multiple parent blocks. In other words, PermitBFT does not establish a path of blocks (a blockchain), but rather a blockDAG (directed acyclic graph).
Table 1: Evaluation of the optimistic latency (Definition 1.1) of selected practical byzantine consensus protocols, plus their message complexity.

| Protocol   | Optimistic latency | Message complexity |
|------------|--------------------|--------------------|
| PBFT [5]   | 3                  | $O(n^2)$           |
| Zyzzya [19]| 4                  | $O(n)$             |
| Tendermint [3] | 3            | $O(n^2)$           |
| SBFT [12]  | 4                  | $O(n)$             |
| HotStuff [27]| 8              | $O(n)$             |
| PaLa [6]   | 4                  | $O(n)$             |
| PermitBFT  | 2                  | $O(n)$             |

*Zyzzya’s speculative latency of 2 rounds may not be achieved with a single crash failure.

**HotStuff employs a pipelining scheme optimizing for throughput rather than latency.

***In practice, the message complexity may be improved to $O(n)$ with the ideas outlined in Section 6.

**Contribution.** As argued above, commit latency is a key efficiency property for BFT protocols; in particular, in a realistic (somewhat optimistic) setting. We use the following optimistic latency model:

**Definition 1.1:** We start the clock when the block creator receives a transaction, and we stop the clock when this transaction is for sure included in the ledger. Local computations are free. The block creator is timely and not byzantine. At most $f$ non-creator nodes are byzantine. Nodes may see messages in a different order, we only assume that messages take at most 1 time unit.

Intuitively, the optimistic latency is designed to describe the time required by a BFT protocol during normal operation, that is, while the network is functioning properly. This is presumably the predominant case in many practical systems. The results are summarized in Table 1.

However, PermitBFT is also fast if it faces a (series of) byzantine block creator(s), as the other nodes will always send a permit to the next creator as soon as they have not received the new block within a timeout. Switching block creators is intrinsic in PermitBFT, so byzantine leaders cannot arbitrarily slow down the system.

Similar to other BFT protocols [5, 27, 26, 3, 6], we demonstrate the safety of the PermitBFT algorithm in the asynchronous model, that is, assuming that messages will be delivered in a finite but unbounded time in arbitrary order. The liveness of the system is demonstrated in the partially synchronous model [9] where synchronous periods (messages being delivered before timeouts occur) may be assumed. For more details, see Sections 4 and 5.1.

**Model.** The PermitBFT algorithm allows $n$ nodes to agree on an ordered log of transactions, establishing a permissioned blockchain. The PermitBFT algorithm tolerates up to $f < \frac{n}{3}$ byzantine failures, that is, malicious adversarial nodes; all other nodes are assumed to be honest nodes following the protocol. For simplicity, we assume that precisely $f$ nodes are byzantine, as byzantine nodes may decide to behave just like honest nodes anyway.

Nodes must be authenticated to participate in the protocol, meaning that all nodes possess a cryptographic public-private key pair suitable for signing messages where the public keys are known to all nodes. As an adversarial model, we assume that byzantine nodes may communicate arbitrarily among each other, thus acting as if they were controlled by a single global adversary with unified information. Byzantine nodes’ computational power, however, is assumed to be polynomially bounded. In particular, we assume that byzantine nodes cannot produce signatures of honest nodes. Cryptographic signatures are the only cryptographic assumption that we make.

## 2 Related Work

The byzantine consensus problem was first posed by Lamport, Shostak and Pease [21] using the metaphor of Byzantine generals who need to cope with possible traitors (byzantine failures) manipulating the generals’ joint decisions.

At first, the byzantine consensus problem was studied in the synchronous setting where multiple solutions were proposed [23, 8, 15, 11]. The lowest latency known for these synchronous solutions is attained by Abraham et al. [11] whose solution requires, in expectation, 8 rounds to reach agreement.

For the asynchronous setting, however, it is known that no deterministic byzantine consensus protocol may solve byzantine consensus in face of a single mere crash failure [10]. In light of this impossibility result for deterministic
protocols, Ben-Or [2] developed a randomized protocol that solves the byzantine consensus problem in the asynchronous setting in expected $O(2^n)$ rounds, observing that all $n$ nodes will eventually obtain the same (or at least a highly biased) outcome for individual, independent coin flips. Subsequently, the latency was lowered to a constant number of expected rounds under the assumption of a random oracle that is available to all nodes [4]. Later, King and Saia [17, 18] proposed a protocol that solves byzantine consensus in expected $O(n^{2.5}\sqrt{\log n})$ rounds in the classic asynchronous model against an adaptive, strong adversary tolerating a mere $f < \frac{n}{400}$ failures.

In an attempt to drive existing theory towards more practical settings, the seminal work of Dwork, Lynch and Stockmeyer [9] introduced the idea to consider the byzantine consensus problem in a partially synchronous setting, that is, to guarantee the safety of a protocol in the asynchronous model while the protocol’s liveness may rely on good networking conditions. By proposing the DLS algorithm [9], the authors also showed the feasibility of byzantine consensus in the partially synchronous setting. The DLS algorithm, however, is still a mostly theoretical result due to its high latency and communication complexity.

The need for a practical BFT protocol was first answered by Castro and Liskov in the form of PBFT [5], a leader-based BFT protocol designed for practical applications. PBFT has been optimized for “normal-case operation”, that is, while the network functions properly (e.g. no message timeouts) and the selected leader is not byzantine. In this case, PBFT incurs a latency of 3 message rounds to reach consensus after receiving a client request. However, in the event of a leader failure, a view change is required. A view change causes both cubic communication overhead and increases the complexity of the PBFT protocol description significantly. It remains to observe that PBFT is widely used in practice, e.g., in the form of BFT-SMaRt [26].

In the following, many proposals were proposed to improve on various properties of PBFT. In particular, Zyzzyva [19] focuses on the idea of a speculative fast track. This can be motivated by the observation that, in practical systems, reliable systems will only rarely experience faulty or malicious behaviour of the participating nodes. In our opinion, however, an optimistic analysis based on the absence of byzantine failures is too optimistic. Almost all proposals (including Zyzzyva itself) motivate the need for a BFT protocol arguing that there is “mounting evidence” [19] for observed byzantine behaviour. Hence, we believe that the BFT property, i.e. the very key property of a byzantine consensus protocol, shall not be neglected in the analysis. Zyzzyva’s speculative fast track attains a latency of 2 communication rounds when there is no byzantine fault whatsoever and the nodes agree on a single client request already (“no contention”). Zyzzyva achieves an optimistic latency (according to Definition 1.1) of 4 communication rounds, as a single mere crash failure suffices to trigger two more rounds of communication.

Another proposal, whilst not related to PBFT, that is entirely focused on a fast track commit in the best case, is Bosco [25]. Bosco claims to achieve a latency of only a single communication round until a transaction can be guaranteed to become committed. However, Bosco requires any-to-any communication in every round ($\Omega(n^2)$ message complexity), tolerates only $f < \frac{n}{2}$ failures and, most importantly, relies on a pre-ordering of the transactions for its fast track. As we argued above, such a scenario is unlikely in systems where transactions may appear concurrently. Consequently, the optimistic latency of Bosco depends on the selected underlying consensus protocol and cannot be compared with the protocols in Table 1.

More recently, Tendermint [3] and SBFT [12] managed to reduce PBFT’s message complexity to a linear number of messages during phases with a correct leader (SBFT: “collectors”). Furthermore, the idea of rotating leaders was introduced in order to avoid the high complexity of the view changes required by PBFT. Their optimistic latencies, however, are 3 and 4 message delays. Note that SBFT’s fast path (with a latency of 2 message rounds) requires that all replicas are non-faulty, including crash failures.

Subsequently, with Ouroboros-BFT [16], HotStuff [27] and PaLa [6], a set of blockchain-based BFT protocols emerged. All of these systems are designed using rotating leaders. Most notably, the authors of HotStuff [27] unified the distinct protocol phases (usually propose, pre-commit and commit) such that each step of commitment for one block can be piggybacked on the previous step of commitment for another block. This concept was adopted by PaLa. Note that the pipeline-setup employed by HotStuff optimizes for throughput rather than immediate latency. Ouroboros-BFT claims a speculative latency of 2 rounds (“instant confirmation”) that does not guarantee the eventual execution of a transaction. Hence, the best optimistic latency is achieved by PaLa with 4 rounds.

Other approaches to increase the scalability of BFT protocols focus on the election of a committee within the set of nodes to drive the byzantine consensus protocol for a specified period of time [11] [14]. In this context, we want to highlight AlgoRand [11], which elects its committees based on verifiable random functions and is shown to be robust against targeted attacks on elected committee members.

PermitBFT follows in the line of blockchain-based BFT protocols with rotating leaders (block creators). However, PermitBFT separates the two powers that a leader typically gets: deciding which transactions to include in a proposed block and deciding where to append a new block, that is, which previous blocks to confirm. By employing a similar
pipelining as HotStuff and PaLa, PermitBFT is on par with these approaches regarding the message complexity incurred during a synchronous phase with a correct node as the block creator.

3 Protocol Description

In PermitBFT, the nodes collectively grant permission to round-robin block creators to append a new block at a specific position in the block graph. If such a block is created (and received by the other nodes) in time, the nodes will permit the next creator to append from this new block only; thus, ensuring that the block graph maintains a specific structure.

The PermitBFT protocol is given as pseudo-code in Algorithm 1. However, we omitted some implementation details. More specifically, nodes collect transactions and fetch unknown blocks (that are referenced by a permit, block or proposal) asynchronously in the background. Furthermore, timeout messages will be answered if a node has already progressed to a newer round (see Section 3.3).

3.1 Optimistic Operation

The block graph is initialized with a genesis block that contains an ordering of the participating nodes. From this ordering, the next block creator is determined in a round-robin manner, specifically as

\[ \text{creator} = \text{round} \mod n. \]

The phase during which a node is the designated creator is loosely referred to as a round. In every round, the nodes collectively grant permission to the next block creator to append a new block at a specific position in the block graph.

**Definition 3.1:** A permit is a tuple \((\text{round}, \text{position})_{\text{issuer}}\) signed by its issuing node.

Intuitively, a permit certifies that a node has endorsed the distinguished creator of round \(\text{round}\) to create a new block at the specified position.

**Definition 3.2:** A proof is a set of \(2f + 1\) permits (from distinct nodes) for the same position and creator.

In other words, a proof guarantees that the creator has been endorsed by a quorum of \(2f + 1\) nodes for the specified position. As there are only \(f\) byzantine nodes, observe that this must include at least \(f + 1\) honest nodes, that is, an honest majority. Hence, a creator that receives \(2f + 1\) permits for the same position is allowed produce a new block.

**Definition 3.3:** A block is a tuple \((\text{proof}, \text{transactions})_{\text{creator}}\) signed by the creator that is determined by the proof.

Consequently, the block graph is established through the position pointers enclosed in the proof of a block. Assuming that byzantine nodes cannot forge the signature of any honest node, observe that a block may only be created by the designated creator itself.

Optimistically, a single round of the protocol proceeds as follows (cf. Figure 1):

1. Whenever a new block is received, nodes issue permits to the next block creator to append from the new block.
2. When the next creator collects a quorum of \(2f + 1\) permits (manages to produce a proof), it creates a new block including all non-conflicting transactions that it received so far.

![Figure 1: Schematic optimistic execution of the PermitBFT algorithm, processing a new request.](image-url)
Algorithm 1: PermitBFT Algorithm

**Input:**
- genesis : initial block of the block graph
- nodes : ordered set of n nodes

**Output:**
block graph with a unique set of committed blocks establishing a totally-ordered transaction log

1: round = 0
2: current = genesis  // current position in the block braid that the node will issue a permit for
3: timeouts = {}  // timeouts is a dictionary mapping a round \(\rightarrow\) set of timeout messages for round

4: repeat
5:   creator = round mod n  // select the round-robin block creator
6:   Send permit(round, current) to the creator  // start round by issuing a permit
7:   if you are the creator:
8:     permits = {}  // permits is a dictionary mapping a position \(\rightarrow\) set of permits for position
9:       repeat
10:         Store newly received permits for this round in the permits dictionary
11:         if there is a position with at least 2f + 1 permits:
12:             proof = set of 2f + 1 permits for position  // we can create a block:
13:                 Send block(proof, position, transactions) to all nodes  // 2. create a new block
14:             break  // 3. end the creator phase
15:         if a creator timeout occurs:
16:             if the permits dictionary contains at least 2f + 1 permits:
17:                 position = minimal position that respects all permits
18:                     Send proposal(position, permits) to all nodes
19:                 break
20:             if received a block, proposal or 2f + 1 timeout messages for any following round:
21:                 break  // must be behind, abort the creator phase
22:       break
23:   if received a result (block or proposal) with result.round \(\geq\) round:
24:     round = result.round  // 1. fast-forward to the latest round
25:     current = result.position  // 2. update current position
26:     break  // 3. stop accepting results for round
27:   if a round timeout occurs:
28:     Send timeout(round) to all nodes  // ensure round synchronization
29:     Store newly received timeout messages in the timeouts dictionary
30:   if there is any following round with at least 2f + 1 timeout messages:
31:     round = maximum of all such rounds
32:     Send 2f + 1 timeout messages for round to all other nodes
33:     break  // end round
34:   round = round + 1
In the best case, this procedure establishes a chain of blocks that may be interpreted as a linear log of transactions. We thus introduce the following commit rule:

**Definition 3.4:** A block is said to be **committed** if there exists at least one child block.

Note that, optimistically, we can thus incorporate a new non-conflicting transaction in a committed block within 2 rounds of communication. In the first round, it may already be incorporated in a new block that is distributed to all nodes by the block creator. In the second round, these nodes issue permits that allow the subsequent block creator to create a child block, which may serve as an *ack* (acknowledgment) message to the client.

**Theorem 3.5:** PermitBFT attains an optimistic latency of 2 communication rounds.

Proof: During a period with synchronous networking conditions with two consecutive honest creators, a new, non-conflicting transaction will be immediately incorporated in a new block that is broadcast to all nodes. Due to synchrony, each honest node will receive the created block before its round timeout expires (see Section 5 on details). In the second round of communication, all honest nodes issue permits specifying the received block as position. Consequently, the subsequent creator may produce a valid proof; hence, a child block for the block containing the new transaction. We will see in **Definition 3.10** that the new transaction may be executed at this point.

Note that we assume transactions to be non-conflicting, which we ensure by employing a UTXO system (see Section 3.4).

### 3.2 Resolving Disagreement

In reality, however, due to the participation of byzantine nodes and possibly unbounded network delays experienced in the asynchronous networking model, a few more scenarios have to be considered. For example, a byzantine creator may not create a block at all, not distribute the created block to all honest nodes, or even create several blocks at the specified position in the block graph using the same proof.

In any case, given the commit rule above that solely relies on the existence of a child block, honest nodes cannot carelessly issue permits for different positions in each round. Honest nodes do, however, maintain a preferred position, their respective current position, which will be updated in case of disagreement. The solution presented by PermitBFT is to allow blocks to be created with multiple parent blocks in such a case.

**Definition 3.6:** A position is a set of blocks in the block graph.

Note that this definition is required to allow blocks with multiple parent blocks, while remaining coherent with the use of the word position in the previous Definitions 3.1 to 3.3. Furthermore, we introduce the following terminology to describe the relations between positions, blocks, permits and proposals in the block graph:

**Definition 3.7:** The depth of a block/position is the length of a shortest path from the genesis block to the block/position.

**Definition 3.8:** We say that a position respects all blocks that lie on any path from the genesis block to the position. Furthermore, a position respects all uncommitted blocks that have depth at most position.depth - 2.

A block/permit/proposal respects all blocks that are respected by the associated position.

A position/permit/proposal is respected by any position that respects all blocks in the (associated) position.

Intuitively, in line with the commit rule, a block respects all its ancestor blocks and blocks that have no chance of ever being committed. Now, for a block creator to demonstrate that there is disagreement among the nodes, we introduce so-called proposals:

**Definition 3.9:** A proposal is a tuple (position, permits) where position is the smallest position that respects all the positions with an issued permit and permits is a set of at least 2f + 1 permits from distinct nodes.

As soon as a node receives a valid proposal, it updates its current position by adopting the associated position of the proposal. In other words, PermitBFT unifies the nodes’ current positions by confirming all positions that may possibly ever become committed by a child block. Note that this is considered a roll-forward (as opposed to a roll-back) approach, where we need to carefully ensure that conflicts will be handled correctly. For that matter, we refer to Section 4

### 3.3 Timeouts & Failures

If possible, an honest block creator should always prefer to create a block over a proposal, as new blocks will ultimately drive progress. However, we can neither guarantee that a block creator will receive the honest nodes’ permit
first, nor that the creator will receive more than \( n - f = 2f + 1 \) permits in any given round due to \( f \) possible byzantine failures. Therefore, PermitBFT employs a creator timeout up to which a creator will wait before potentially issuing and broadcasting a proposal.

On the other end, from the perspective of nodes that are not the designated creator of the round, it may be difficult to determine the end of a round. To that end, a (longer) round timeout is used. In order to guarantee sufficient synchronization among the nodes, each node broadcasts a timeout message upon expiry of their local round timeout. A collection of \( 2f + 1 \) timeout messages is required to proceed to the next round without observing either a block or a proposal from the designated creator of this or any following round.

Note that a byzantine creator or bad networking conditions may cause some honest node to incur a local round timeout, while other honest nodes did receive a block or proposal for the round. Thus, if an honest node receives a timeout message for an “old round” (any round number smaller than its local round variable), the honest node answers by relaying the latest block, proposal, or \( 2f + 1 \) timeout messages that it has seen locally. Note that this has been omitted in Algorithm 1 for simplicity.

3.4 Inferring the Transaction Log

Similar to the Bitcoin protocol [22], we assume an “Unspent Transaction Output” (UTXO) [7] scheme which allows us to assume that conflicting transactions may only occur in case of malicious behavior. Note that two conflicting transactions may be included in different blocks that could become committed simultaneously (but not independently, see Section 4.2). To that end, let us define when a transaction itself is considered to be committed:

**Definition 3.10:** A transaction is said to be committed if it is contained in a committed block \( b \) and there exists a child block \( b' \) (a witness of the commitment of \( b \)) such that there are no conflicting transactions within the set of committed blocks that are respected by the child block \( b' \).

If any two transactions are conflicting, we cannot guarantee that none of these transactions have been committed by a (possibly hidden) block. Hence, we can neither choose a single one of the transactions to be executed, nor can we drop the conflicting transactions entirely. Consequently, using the fact that conflicting transactions may only occur in case of malicious behavior, we freeze the transactions. In the context of a cryptocurrency, for instance, this would imply locking the maximum amount spent across the set of pairwise conflicting transactions. This is necessary in order to guarantee that a committed transaction may be executed once its proof of commitment, a child block only respecting a single one of the conflicting transactions, is revealed. In the worst case, this may lead to deadlocked funds; hence, immediately penalizing the transaction owner who acted maliciously.

Note that the set of blocks that are respected by a block at depth \( d \) is immutable. To see this, observe that a majority of honest nodes must have issued permits for a position at depth \( d - 1 \) in order to create the block. Hence, no new blocks can be created at depths smaller than \( d - 1 \).

**Corollary 3.11:** Once a transaction is committed, it is irrevocably committed.

Hence, as soon as a transaction is committed, it may be safely executed (see Section 4.2 for details). So, how do we infer a linear, totally-ordered log of transactions from the resulting block graph? PermitBFT includes transactions into the ledger in two steps. At first, it is ensured that no two conflicting transactions become committed simultaneously which makes them safe for execution. In a second step, the definitive ordering in the ledger is finalized.

In other words, PermitBFT creates a totally-ordered ledger with a set of unordered, non-conflicting committed transactions at its head. These transactions are yet to be put into their definitive order but can already be executed.

4 Safety

As for every distributed system, it is crucial to guarantee the safety of the PermitBFT algorithm independent of both the state of the system and any networking assumptions (i.e. in the asynchronous model). By safety of the PermitBFT algorithm, we denote that two conflicting transactions may not become committed simultaneously. To that end, we first analyze the structural properties of the block graph and show that there cannot be any two conflicting but independently committed blocks. Subsequently, in Section 4.2, we demonstrate that this property ensures the transaction safety of the PermitBFT algorithm.

Formally, to accommodate for arbitrary networking conditions, we base our analysis on the asynchronous model throughout this section. That is, we assume that messages will be delivered within a finite but unbounded time, without message loss. In particular, the order in which messages are received may be arbitrary (even from the same sender).

**Remark:** In the following, line numbers always refer to the respective lines in Algorithm 1 for simplicity.
4.1 Safety of the Block Graph

Intuitively, the safety of the PermitBFT algorithm is maintained by the honest nodes who do not switch their respective current position carelessly. Each node, individually, will decide on one of four possible results for each round before progressing to any following round: either accepting a block or a proposal (line 25), skipping to the next round with a timeout (line 31), or by fast-forwarding to a following round (lines 24 and 31).

We thus show that a node may safely accept a block were created.

We thus show that a node may safely accept a block were created. Let round \( i \) be the first round when such a child block was created. Thus, there must have been \( 2f + 1 \) permits for a particular position containing the block in round \( i \). Given that at most \( f \) of these permits may be issued by byzantine nodes, at least \( f + 1 \) honest nodes must have issued a permit for the same particular position in round \( i \). Hence, the block is promised. \( \square \)

In Appendix A we show that the PermitBFT algorithm maintains the following safety invariant:

**Definition 4.1:** A block is said to be promised, if in any round up to (and including) the latest round, at least \( f + 1 \) honest nodes issued a permit for a particular position containing the block.

**Lemma 4.2:** If a block is committed, it is also promised.

Proof: To commit a block, a child block must be created. Let round \( i \) be the first round when such a child block was created. Thus, there must have been \( 2f + 1 \) permits for a particular position containing the block in round \( i \). Given that at most \( f \) of these permits may be issued by byzantine nodes, at least \( f + 1 \) honest nodes must have issued a permit for the same particular position in round \( i \). Hence, the block is promised. \( \square \)

**Theorem 4.4:** In any round, a majority of honest nodes would only issue a safe permit.

In the following, we demonstrate what implications Theorem 4.4 imposes on the set of promised blocks:

**Definition 4.5:** A safe permit is a permit for a position respecting all promised blocks.

**Theorem 4.4:** In any round, a majority of honest nodes would only issue a safe permit.

In the following, we demonstrate what implications Theorem 4.4 imposes on the set of promised blocks:

**Definition 4.6:** We say that two blocks \( b_1 \) and \( b_2 \) are conflicting if neither \( b_1 \) respects \( b_2 \), nor \( b_2 \) respects \( b_1 \).

**Definition 4.7:** Two conflicting blocks cannot become committed independently.

Proof: We prove the statement by contradiction. Without loss of generality, assume that block \( b_1 \) becomes promised in round \( i \) and block \( b_j \) in round \( j \), where \( i \leq j \). By Theorem 4.4 however, at least \( f + 1 \) honest nodes would only cast a safe permit in round \( j \). Thus, at least \( f + 1 \) honest nodes would issue a permit for a position respecting block \( b_j \) in round \( j \). As there are only \( 2f + 1 \) honest nodes, there are at most \( 2f + 1 - (f + 1) = f \) honest nodes who issue a permit for a position respecting the block \( b_j \), but not the block \( b_i \), which is not sufficient for the block \( b_j \) to become promised in round \( j \). This contradicts the assumption that two independently promised blocks exist. \( \square \)

The pairwise application of Lemma 4.7 on a set of conflicting blocks in combination with Lemma 4.2 yields the following corollary.

**Corollary 4.8:** Out of a set of conflicting blocks, only one block may be independently committed by a child block.

4.2 Transactions with Conflicts

Recall that each block comes with a set of transactions. As outlined in Section 3.4, we assume a UTXO system where conflicting transactions are, without exceptions, proof of malicious behaviour.

Following Definition 3.10, a transaction is said to be committed if it is contained in a committed block and there exists a child block such that there are no conflicting transactions within the set of committed blocks that are respected by the child block.

We establish the transaction safety property mentioned above and thereby guarantee that double-spending is impossible. To that end, we argue for the given transaction commit rule that it applies to at most one transaction out of a set of conflicting transactions.

**Theorem 4.9 (safety):** Two conflicting transactions cannot become committed simultaneously.
Proof: To begin with, note that two conflicting transactions may only become committed simultaneously as part of independently committed blocks. In other words, two conflicting transactions cannot be committed jointly within a single block or within a set of blocks that become committed as a position together. To see this, recall that the commit rule explicitly requires that there is no conflicting transaction within the set of respected blocks of some child block.

We assume for contradiction that two conflicting transactions may become committed simultaneously within independently committed blocks. Hence, either the blocks are conflicting or not. If the blocks are conflicting, Corollary 4.8 guarantees that at most one such conflicting block may be committed independently. If the blocks are not conflicting, then one of the blocks respects the other and thus the transaction commit rule only applies to the transaction within the respected block. □

5 Liveness

By liveness, we denote the property that a distributed system is guaranteed to make progress as soon as the network reaches a state satisfying certain timing assumptions. More specifically, to make progress means that outstanding transactions will be included in the ledger and irrevocably committed.

5.1 Timing

First, note that we have previously analyzed the safety of the PermitBFT algorithm without the need to consider any timing information (i.e. in the asynchronous model). For PermitBFT’s liveness, however, timing is important. For the purpose of this analysis, we assume a global perspective with full information about the state of the entire system at any considered time $t$.

Throughout this section, we assume the partially synchronous model [9], that is, we assume that phases of synchronous operation do occur. During a phase of synchronous operation, we assume that messages among honest nodes will be received within a known maximum message delay $\Delta$. Furthermore, we assume that local timers can reliably count the maximum message delays $\Delta$. In particular, we assume that $2\Delta < \text{creator timeout} < 3\Delta$ and $5\Delta < \text{round timeout}$ (5.1) hold reliably for all honest nodes, where creator timeout and round timeout are the actual times for the respective local timeouts to occur. Local computation is considered to be negligible.

5.2 Liveness Guarantees

In Appendix B, we demonstrate the following basic liveness guarantee for the PermitBFT algorithm:

**Theorem 5.2:** Assuming that sufficiently long periods of synchrony do occur, the PermitBFT algorithm will create and append new blocks to the ledger.

At this point, it remains to conclude that new transactions will eventually be included in the ledger and irrevocably committed.

**Theorem 5.3 (liveness):** A transaction without conflicts will eventually become executed by all honest nodes.

Proof: An honest creator includes all collected non-conflicting transactions in any block it creates (see Section 3.1). Hence, following [Theorem 5.2] a non-conflicting transaction will be included in a block during a sufficiently long phase of synchronous operation. Furthermore, the created block will be received and accepted by all honest nodes before their round timeouts expire. All honest nodes will subsequently issue permits to append new blocks from the block containing the non-conflicting transaction. Following [Lemma A.2] and [Theorem 4.4], all future blocks will thus extend from the block containing the non-conflicting transaction. Upon another period of synchronous operation (possibly within the same period), three consecutive honest creators will create another block by [Theorem 5.2]. Thereby, the commit rule [Definition 3.4] is satisfied for the non-conflicting transaction and it will be executed by all honest nodes upon receiving the new block. □

6 Message Complexity

Optimistically, the PermitBFT algorithm requires the nodes to send $O(n)$ messages to create a block (see Figure 1). Note that the message lengths can be optimized using threshold cryptography (as suggested in [12, 27]).

In both the cases of a byzantine block creator or a network failure, however, the nodes fall back to a round of any-to-any communication for a synchronization step, incurring a message complexity of $\Theta(n^2)$. Observe that this inflated
communication pattern is mainly used in order to tolerate even a network split over an unbounded period of time. Such extreme network conditions should be very rare in practice. We thus suggest that such drastic measures may be employed only every \( n \) rounds, while keeping the other rounds synchronized based on the local clocks. We thus expect a resulting communication complexity of \( O(n) \) messages.

Note that in signal processing, e.g. GPS localization, low-drift oscillators with \( \pm 0.5 \text{ ppm} \) (less than 0.0001 \% or 1 \( \mu \text{s} \)) are common [13]. Thus, even with hundreds of nodes in the system, it should be possible to maintain a sequence of unified rounds over a course of \( n \) rounds by increasing the maximum message delay by a small factor in each round after a synchronization step. The careful analysis and implementation of such optimizations, however, lies outside the scope of this paper and is considered future work.

References

[1] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Efficient synchronous byzantine consensus. arXiv preprint arXiv:1704.02397, 2017.

[2] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In Proceedings of the second annual ACM symposium on Principles of distributed computing, pages 27–30. ACM, 1983.

[3] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. arXiv preprint arXiv:1807.04938, 2018.

[4] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. Journal of Cryptology, 18(3):219–246, 2005.

[5] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In OSDI, volume 99, pages 173–186, 1999.

[6] T. H. Chan, R. Pass, and E. Shi. Pala: A simple partially synchronous blockchain. Technical report, Cryptology ePrint Archive, Report 2018/981, 2018. https://eprint.iacr.org, 2018.

[7] S. Delgado-Segura, C. Pérez-Sola, G. Navarro-Arribas, and J. Herrera-Joancomartí. Analysis of the bitcoin UTXO set. In International Conference on Financial Cryptography and Data Security, pages 78–91. Springer, 2018.

[8] D. Dolev and H. R. Strong. Polynomial algorithms for multiple processor agreement. In Proceedings of the fourteenth annual ACM symposium on Theory of computing, pages 401–407. ACM, 1982.

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

[10] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process, 1985.

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

[12] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A scalable decentralized trust infrastructure for blockchains. arXiv preprint arXiv:1804.01626, 2018.

[13] IQD Frequency Products. IQD TCXO Specification, 2018. https://www.iqdfrequencyproducts.com/products/pn/LFTCXO077227Reel.pdf.

[14] M. M. Jalalzai, C. Busch, and G. Richard III. Proteus: A scalable BFT consensus protocol for blockchains. arXiv preprint arXiv:1903.04134, 2019.

[15] J. Katz and C.-Y. Koo. On expected constant-round protocols for byzantine agreement. Journal of Computer and System Sciences, 75(2):91–112, 2009.

[16] A. Kiayias and A. Russell. Ouroboros-BFT: A simple byzantine fault tolerant consensus protocol. Technical report, Cryptology ePrint Archive, Report 2018/1049, 2018. https://eprint.iacr.org, 2018.

[17] V. King and J. Saia. Byzantine agreement in expected polynomial time. Journal of the ACM (JACM), 63(2):13, 2016.

[18] V. King and J. Saia. Correction to byzantine agreement in expected polynomial time, JACM 2016. arXiv preprint arXiv:1812.10169, 2018.

[19] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In ACM SIGOPS Operating Systems Review, volume 41, pages 45–58. ACM, 2007.
[20] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

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

[22] S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

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

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

[25] Y. J. Song and R. van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

[26] J. Sousa, E. Alchieri, and A. Bessani. State machine replication for the masses with BFT-SMaRt. 2013.

[27] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.
Appendix A: Proof of the Safety Invariant

To begin with, recall [Definition 4.3]

Definition 4.3: A safe permit is a permit for a position respecting all promised blocks.

Furthermore, we define:

Definition A.1: A round is called safe if at least \( f + 1 \) honest nodes would only cast a safe permit.

Note that a safe round is defined based on the potentially issued permits of honest nodes. This way, we can reason about rounds in which the network synchronization does not allow us to assume that an honest node will actually issue a permit for some round \( i \). However, to produce a valid block or proposal, any creator node must have received sufficiently many, that is, at least \( 2f + 1 \) permits (whereof at least \( f + 1 \) must be honest permits).

A.1 Safety of Blocks

To begin with, we consider the main backbone of the PermitBFT algorithm: the blocks.

Lemma A.2: If round \( i \) is safe, a block created in round \( i \) respects all promised blocks.

Proof: A block must be created with \( 2f + 1 \) permits from the same round \( i \) for the exact same position \( p \) (line 11). Hence, there must have been at least \( f + 1 \) honest permits for position \( p \) in round \( i \). Given that at least \( f + 1 \) honest nodes would have only issued a safe permit in round \( i \), we conclude that position \( p \) respects all promised blocks.

Furthermore, we show that the safety invariant is maintained if at least \( f + 1 \) honest nodes accept the same block.

Lemma A.3: If round \( i \) is safe and a block is accepted by at least \( f + 1 \) honest nodes, then round \( i + 1 \) is safe.

Proof: If there is a block \( b \) created in round \( i \) that is accepted by at least \( f + 1 \) honest nodes, then block \( b \) will become promised in round \( i + 1 \). Since at least \( f + 1 \) honest nodes have accepted the block, however, at least \( f + 1 \) honest nodes will cast a permit for the block \( b \) in round \( i + 1 \). Note that these are safe permits as there is at most one block created in round \( i \) that may be accepted by at least \( f + 1 \) honest nodes as there are only \( 2f + 1 \) honest nodes. Similarly, no proposal may be issued and accepted by at least \( f + 1 \) honest nodes in round \( i \).

A.2 Safety of Proposals

Next, let us have a closer look at the role of a proposal in the PermitBFT algorithm. In essence, in a safe round, a proposal is a certificate that no promised block may be excluded from the proposal’s position. Hence, intuitively, we show that it is always safe for any honest node to accept a proposal.

As a preliminary step, we show that a proposal’s position respects all blocks that could have become promised so far, assuming that the honest nodes properly maintained the safety invariant of the PermitBFT algorithm thus far.

Lemma A.4: If round \( i \) is safe, a proposal in round \( i \) respects all promised blocks.

Proof: A proposal must include \( 2f + 1 \) permits. At most \( n - (2f + 1) = f \) honest nodes’ permits could have been excluded from the proposal. Hence, the proposal must include at least one of the \( f + 1 \) honest nodes’ safe permits and thus respect all promised blocks.

Subsequently, we show that the safety invariant is maintained if at least \( f + 1 \) honest nodes accept the same proposal.

Lemma A.5: If round \( i \) is safe and a proposal is accepted by at least \( f + 1 \) honest nodes, then round \( i + 1 \) is safe.

Remark: For simplicity we assume that an honest node that does not actively see a proposal, but for arbitrary reasons will issue a permit for the same position in round \( i + 1 \), is also considered to have accepted the proposal (implicitly).

Proof: If there is a proposal issued in round \( i \) that is accepted by at least \( f + 1 \) honest nodes, the block(s) at the associated position \( p \) will become promised latest in round \( i + 1 \). Since at least \( f + 1 \) honest nodes have accepted the proposal, however, at least \( f + 1 \) honest nodes will issue a permit for the position \( p \) in round \( i + 1 \). Note that the position \( p \) is safe as it respects all previously promised blocks (Lemma A.4) and there is at most one proposal issued in round \( i \) that may be accepted by at least \( f + 1 \) honest nodes as there are only \( 2f + 1 \) honest nodes. Similarly, no block may be created and accepted by at least \( f + 1 \) honest nodes in round \( i \).

A.3 Safety of other Rounds

Finally, let us consider the case when no block or proposal is accepted by at least \( f + 1 \) honest nodes in some round. To begin with, we observe that no new block may become promised in the subsequent round:
**Lemma A.6:** If round $i$ is safe and there is no block or proposal that is accepted by at least $f + 1$ honest nodes, then no new block may become promised by the permits in round $i + 1$.

Proof: We prove the lemma by demonstrating that there is no position receiving at least $f + 1$ honest permits for the first time in round $i + 1$. Any new block is accepted by less than $f + 1$ honest nodes and there cannot be any other honest permit for such new blocks in round $i + 1$. Otherwise, honest nodes only update their current position according to a received proposal. Assuming that there is no proposal that is accepted by at least $f + 1$ honest nodes, that is, whose position is adopted by at least $f + 1$ honest nodes, there cannot be a position receiving at least $f + 1$ honest permits. □

Thus, it remains to show that all the actions taken by honest nodes in such a round will maintain the safety invariant.

**Lemma A.7:** If round $i$ is safe and there is no block or proposal that is accepted by at least $f + 1$ honest nodes, then round $i + 1$ is safe.

Proof: Following Lemma A.6, it remains to show that at least $f + 1$ honest nodes’ permits remain safe permits. Without loss of generality, let $v$ be one of the $f + 1$ honest nodes who cast a safe permit in round $i$.

On the one hand, observe that each honest node transitioning to the next round without updating its current position would only cast a permit for the same position again. In other words, $v$ would only cast a safe permit as its permit still respects all previously promised blocks and there are no newly promised blocks in round $i + 1$.

On the other hand, an honest node would only update its current position by accepting a block or a proposal (line 25). In any case, the updated current position of an honest node respects all previously promised blocks (Lemmas A.2 and A.4). Hence, as there are no newly promised blocks in round $i + 1$, we conclude that the honest node $v$ would only cast a safe permit for round $i + 1$. □

### A.4 Safety Invariant

Ultimately, we may combine the previous statements to conclude Theorem 4.4, namely that every round is a safe round:

**Theorem 4.4:** In any round, a majority of honest nodes would only issue a safe permit.

Proof: We show this statement by induction on the round number $i$. For $i = 0$, observe that the statement holds as all honest nodes start round 0 by voting for the genesis block, which is also the only block that may become promised in this round. From now on, assume that some round $i \geq 0$ is safe. We thus have to show that round $i + 1$ is also safe.

If there is a block or proposal that is accepted by at least $f + 1$ honest nodes, then round $i + 1$ is safe by Lemmas A.3 and A.5. Otherwise, Lemma A.7 yields the desired statement. □

In other words, in every round, at least $f + 1$ honest nodes’ permits ensure that all promised blocks are contained in a newly promised position.
Appendix B: Proof of the Basic Liveness Guarantee

To show a basic liveness guarantee, we show that three consecutive honest creators are guaranteed to make progress during a phase of synchrony. Subsequently, we argue that there must exist at least one such constellation with three consecutive honest creators and deduce a basic liveness guarantee in the form of Theorem 5.2.

B.1 Round Synchronization

To begin with, we introduce some terminology to describe the possible states of the system over time.

**Definition B.1:** A node \( u \) is said to be in round \( r_u(t) \) at time \( t \), that is, the value of its round variable at time \( t \). Furthermore, let
\[
    r_{\text{max}}(t) := \max_{u \in \text{honest}} r_u(t)
\]
be the maximum round of any honest node.

**Definition B.2:** A round \( i \) is started, once the first honest node has started the execution of round \( i \). More formally, round \( i \) is started at time \( T_i \), where
\[
    T_i := \min\{ t \mid r_{\text{max}}(t) \geq i \}.
\]

We say that a round \( i \) is unified, if all honest nodes start the execution of that round in the interval \( [T_i, T_i + \Delta] \).

Next, we demonstrate that any honest creator will sufficiently synchronize the round numbers of all honest nodes.

**Lemma B.3:** Assuming synchrony, starting a round \( i \) with an honest creator \( u \) implies that round \( i + 1 \) is unified.

Proof: Observe that round \( i = r_{\text{max}}(t) \). Round \( i + 1 \) starts at time \( T_{i+1} \) when the first honest node \( v \) receives a block or a proposal (line 23) or \( 2f + 1 \) timeout messages (line 30) for round \( i \).

Suppose that \( v \) has received a block or proposal first. Given that the creator \( u \) is honest and the only node capable of producing a block or proposal in round \( i \), \( u \) must have created the block or proposal. As \( u \) is honest, it must also be the first honest node to receive the created block or proposal (without network delay), so \( v = u \). This immediately determines \( T_{i+1} \).

Otherwise, suppose that \( v \) receives \( 2f + 1 \) timeout messages first. Again, \( v \) immediately transitions to round \( i + 1 \) and thus determines \( T_{i+1} \).

In any case, \( v \) will immediately broadcast the created block or proposal (lines 13 and 18) or the \( 2f + 1 \) collected timeout messages (line 32) to all nodes at time \( T_{i+1} \). Due to synchrony, each honest node is guaranteed to receive the block, proposal or \( 2f + 1 \) timeout messages within the message delay \( \Delta \) and will immediately transition to round \( i + 1 \), no later than time \( T_{i+1} + \Delta \). \( \square \)

Note that not every honest creator \( u \) necessarily reaches round \( i \) at all. In the case that \( u \) does not ever reach round \( i \), round \( i + 1 \) would be started as soon as some other honest node \( v \) receives \( 2f + 1 \) timeout messages for round \( i \).

B.2 Making Progress

Subsequently, we show that an honest creator of a unified round will always create a block or issue a proposal during a phase of synchronous operation, independent of the honest nodes’ current positions.

**Lemma B.4:** Assuming synchrony, an honest creator \( u \) will create a block or issue a proposal in a unified round \( i \).

Proof: In a unified round \( i \), all honest nodes start the execution of the round within the interval \( [T_i, T_i + \Delta] \). Hence, the honest nodes will send their permit to the creator \( u \) (line 6) for round \( i \) latest at time \( T_i + \Delta \). Assuming synchrony, the honest creator \( u \) is thus guaranteed to have received at least \( 2f + 1 \) permits at time \( T_i + 2\Delta \).

Given that the creator \( u \) did not begin round \( i \) before the time \( T_i \), it must have started its creator timeout earliest at time \( T_i \). Consequently, the creator \( u \) receives all permits that arrive until \( T_i + \text{creator timeout} \). Following Equation 5.1 we have
\[
    T_i + 2\Delta < T_i + \text{creator timeout}.
\]

Thus, by the time the creator timeout expires, the creator \( u \) will either have created a block already, or issue a proposal now as it received at least \( 2f + 1 \) permits from the honest nodes. \( \square \)

If all honest nodes issue permits for the same position, we observe that an honest creator will create a block.

**Corollary B.5:** Assuming synchrony, an honest creator \( u \) will create a block in a unified round \( i \) if all honest nodes issue permits for the same position.
In order to guarantee progress on the ledger, we confirm that any block or proposal created by an honest creator in a unified round will be accepted by all honest nodes before each of them transitions to the next round.

**Lemma B.6:** Assuming synchrony, a block or proposal created by an honest creator in a unified round $i$ will be received and accepted as the new current position by all honest nodes before starting to execute the next round $i + 1$.

Proof: Recall from the proof of [Lemma B.4](#) that an honest creator $u$ of a unified round $i$ will have created a block or issued a proposal (latest) by the time its creator timeout expires.

Given that round $i$ is a unified round, the creator $u$ has started its creator timeout latest at time $T_i + \Delta$. Consequently, the creator timeout is guaranteed to expire before $T_i + 4\Delta$ (following [Equation 5.1](#)). Now, the honest creator $u$ immediately sends the created block or proposal to all nodes (lines 13 and 18). All honest nodes thus receive the block or proposal latest at time $T_i + 5\Delta$ due to the assumed synchrony of the network among honest nodes.

By definition of $T_i$, we know that no honest node has started their local round timeout before time $T_i$. Following [Equation 5.1](#) we have

$$T_i + 5\Delta < \text{round timeout}.$$ 

Consequently, no honest node’s round timeout has expired at time $T_i + 5\Delta$ and, therefore, no honest node would have created a timeout message at this point. Since the byzantine nodes can create at most $f$ timeout messages, no honest node would have received $2f + 1$ timeout messages. Thus, all honest nodes will receive and accept the created block or proposal (line 25) before starting to execute the next round $i + 1$.

Note that an honest node would fast-forward to a round $j + 1 > i$, if it received a block, a proposal, or $2f + 1$ timeout messages for round $j$ (lines 24 and 31). On the one hand, no such round $j \geq i$ could start before round $i$ (i.e. $T_j \geq T_i$); hence, no honest node would have created a timeout message for round $j$ by the time $T_i + 5\Delta$.

On the other hand, any block or proposal for round $j - 1$ would have to be created using the permits received from at least $f + 1$ honest nodes. However, given that there cannot exist sufficiently many timeout messages to abort round $i$, these $f + 1$ honest nodes must have already received and accepted the block or proposal of round $i$. In consequence, a block or proposal created in a following round must either extend from the block created in round $i$, or respect the position of the proposal issued in round $i$, respectively. In any case, an honest node accepting such a block or proposal to fast-forward to round $j + 1$ instead, will implicitly accept the block or proposal of round $i$. \hfill $\square$

Before we establish the basic liveness guarantee for the PermitBFT algorithm, we observe one final property:

**Lemma B.7:** There exists at least one sequence of three consecutive honest creators within $n + 2$ consecutive rounds.

Proof: We have $n = 3f + 1$ nodes, whereof $f < \frac{n}{3}$ are byzantine. Recall that we choose the creator of a round in a round-robin (wrap-around) scheme. By arranging the nodes in a circle, we may pick the first of the three consecutive honest creators among the next $n$ nodes. The third consecutive honest creator will be reached within at most $n + 2$ rounds. It remains to show that one iteration of the circle contains three consecutive honest creators.

To that end, consider any fixed order of the $2f + 1$ honest creators in a circle. The honest creators can be divided into sequences of consecutive honest creators by placing byzantine nodes in between. Note that, by placing the $f$ byzantine nodes, the honest creators can be partitioned into at most $f$ sequences. Now, if we assume for contradiction that each of these sequences of consecutive creators would have length at most 2, then there could only be $2f < 2f + 1$ honest nodes – a contradiction. \hfill $\square$

Ultimately, we may now conclude [Theorem 5.2](#)

**Theorem 5.2:** Assuming that sufficiently long periods of synchrony do occur, the PermitBFT algorithm will create and append new blocks to the ledger.

Proof: Once a sufficiently long period of synchrony occurs, it follows from [Lemma B.7](#) that there must be three consecutive rounds $r_1$, $r_2$, and $r_3$ with honest creators within the next $n + 2$ rounds. A round can be terminated either by creating a block or proposal, or with $2f + 1$ timeout messages. Hence, given that there are $2f + 1$ honest nodes, the round $r_1$ will be started at some point.

By [Lemma B.3](#) the rounds $r_2$ and $r_3$ are unified rounds. Furthermore, by [Lemmas B.4 and B.6](#) the creator of round $r_2$ creates a block or issues a proposal that will be received (and accepted as the new current position) by each honest node before starting to execute the next round $r_3$. Note that all honest nodes will now issue permits for the same position in round $r_3$. Consequently, by [Corollary B.5](#) and [Lemma B.6](#), the creator of round $r_3$ is guaranteed to create a block that is received and accepted by all honest nodes before starting the next round. \hfill $\square$