Internet Computer Consensus

Jan Camenisch, Manu Drijvers, Timo Hanke,
Yvonne-Anne Pignolet, Victor Shoup, Dominic Williams

DFINITY Foundation

May 13, 2021

Abstract

We present the Internet Computer Consensus (ICC) family of protocols for atomic broadcast (a.k.a., consensus), which underpin the Byzantine fault-tolerant replicated state machines of the Internet Computer. The ICC protocols are leader-based protocols that assume partial synchrony, and that are fully integrated with a blockchain. The leader changes probabilistically in every round. These protocols are extremely simple and robust: in any round where the leader is corrupt (which itself happens with probability less than 1/3), each ICC protocol will effectively allow another party to take over as leader for that round, with very little fuss, to move the protocol forward to the next round in a timely fashion. Unlike in many other protocols, there are no complicated subprotocols (such as "view change" in PBFT) or unspecified subprotocols (such as "pacemaker" in HotStuff). Moreover, unlike in many other protocols (such as PBFT and HotStuff), the task of reliably disseminating the blocks to all parties is an integral part the protocol, and not left to some other unspecified subprotocol. An additional property enjoyed by the ICC protocols (just like PBFT and HotStuff, and unlike others, such as Tendermint) is *optimistic responsiveness*, which means that when the leader is honest, the protocol will proceed at the pace of the actual network delay, rather than some upper bound on the network delay. We present three different protocols (along with various minor variations on each). One of these protocols (ICC1) is designed to be integrated with a peer-to-peer gossip sub-layer, which reduces the bottleneck created at the leader for disseminating large blocks, a problem that all leader-based protocols, like PBFT and HotStuff, must address, but typically do not. Our Protocol ICC2 addresses the same problem by substituting a low-communication reliable broadcast subprotocol (which may be of independent interest) for the gossip sub-layer.

1 Introduction

Byzantine fault tolerance (BFT) is the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of some of its components while still functioning properly as a whole. One approach to achieving BFT is via *state machine replication* [Sch90]: the logic of the system is replicated across a number of machines, each of which maintains state, and updates its state is by executing a sequence of *commands*. In order to ensure that the non-faulty machines end up in the same state, they must each deterministically execute the same sequence of commands. This is achieved by using a protocol for *atomic broadcast*.

In an atomic broadcast protocol, we have n parties, some of which are honest (and follow the protocol), and some of which are corrupt (and may behave arbitrarily).

Roughly speaking, such an atomic broadcast protocol allows the honest parties to schedule a sequence of *commands* in a consistent way, so that each honest party schedules the same commands in the same order.

Each party receives various commands as input — these inputs are received incrementally over time, not all at once. It may be required that a command satisfy some type of validity condition, which can be verified locally by each party. These details are application specific and will not be further discussed.

Each party outputs an ordered sequence of commands — these outputs are generated incrementally, not all at once.

One key security property of any secure atomic broadcast protocol is **safety**, which means that each party outputs the *same* sequence of commands. Note that at any given point in time, one party may be further along in the protocol than another, so this condition means that at any point in time, if one party has output a sequence s and another has output a sequence s', then s must be a prefix of s', or vice versa.

Another key property of any secure atomic broadcast protocol is **liveness**. There are different notions of liveness one can consider. In one notion, the requirement is that each honest party's output queue grows over time at a "reasonable rate" (relative to the speed of the network). This notion of liveness is quite weak, in that it does not rule out the possibility of some parties having their input commands ignored indefinitely. In another, stronger notion of liveness, the requirement is that if "sufficiently many" parties receive a particular command as input at some point in time, then that command will appear in the output queues of all honest parties "not too much later". Of course, even this definition is incomplete without precisely defining "sufficiently many" and "not too much later".

The Internet Computer Consensus (ICC) family of protocols. In this paper, we present a family of atomic broadcast protocols which correspond to the atomic broadcast protocol used in the Internet Computer [DFI20]. To a first approximation, the Internet Computer is a dynamic collection of intercommunicating replicated state machines: commands for atomic broadcast on one replicated state machine are either derived from messages received other replicated state machines, or from external clients. We actually present three specific protocols, ICC0, ICC1, and ICC2. Protocol ICC0 is a somewhat simplified version of the protocol actually used in the Internet Computer, but is easier to present and to analyze, and it is the main focus of most of this paper. Protocol ICC1 most closely models the version of the protocol used in the Internet Computer, and is only slightly more involved than ICC0. Protocol ICC2 goes a bit beyond ICC1, and uses techniques that are not currently used in the Internet Computer. We emphasize that the ICC protocols are *fully specified* (they do not rely on unspecified, non-standard components), *extremely simple* (a fairly detailed description easily fits on a single page), and *robust* (performance degrades gracefully in the face of Byzantine attack).

In designing and analyzing any protocol for atomic broadcast, certain assumptions about the nature and number of corrupt parties and the reliability of the network are critical. We will assume throughout this paper that at most $t < n/3$ of the parties are corrupt, and may behave arbitrarily and are completely coordinated by an adversary. This includes, of course, parties that have simply "crashed". We do, however, assume that the adversary chooses which parties to corrupt *statically*, at the beginning of the execution of the protocol.

Regarding the network, there are a few different assumptions that are typically made:

- At one extreme, one can assume that the network is *synchronous*, which means that all messages sent from an honest party to an honest party arrive within a known time bound Δ_{bnd}.
- At the other extreme, one can assume that the network is *asynchronous*, meaning that messages can be arbitrarily delayed.

In between these two extremes, various *partial synchrony* assumptions can be made [DLS88]. For our analysis here, the type of partial synchrony assumption we shall need is that the network is synchronous for relatively short intervals of time every now and then.

Regardless of whether we are assuming an asynchronous or partially synchronous network, we will assume that every message sent from one honest party to another will *eventually* be delivered.

Like a number of atomic broadcast protocols, each of the ICC protocols is *blockchain* based. As the protocol progresses, a tree of blocks is grown, starting from a special "genesis block" that is the root of the tree. Each non-genesis block in the tree contains (among other things) a *payload*, consisting of a sequence of commands, and a hash of the block's parent in the tree. The honest parties have a consistent view of this tree: while each party may have a different, partial view of this tree, all the parties have a view of the *same* tree. In addition, as the protocol progresses, there is always a path of *committed* blocks in this tree. Again, the honest parties have a consistent view of this path: while each party may have a different, partial view of this path, all the parties have a view of the *same* path. The commands in the payloads of the blocks along this path are the commands that are output by each party.

The protocol proceeds in *rounds*. In the kth round of the protocol, one or more depth-k blocks are added to the tree. That is, the blocks added in round k are always at a distance of exactly k from the root. In each round, a *random beacon* is used to generate a random permutation of the n parties, so as to assign to each party a *rank*. The party of lowest rank is the *leader* of that round. When the leader is honest and the network is synchronous, the leader will propose a block which will be added to the tree. If the leader is not honest or the network is asynchronous, some other parties of higher rank may also propose blocks, and also have their blocks added to the tree. In any case, the logic of the protocol gives highest priority to the leader's proposed block.

We show that:

- Each of the ICC protocols provides liveness under such a partial synchrony assumption. Very roughly speaking, whenever the network remains synchronous for a short while, whatever round the parties are in at that time, if the leader is honest, only the leader's block will be added to the tree of blocks at that round, and all the nodes along the path from the root to that block will be committed.
- Each of the ICC protocols provides safety, even in the asynchronous setting.

In the most basic version of the ICC protocols, the communication-delay bound Δ_{bnd} in the partial synchrony assumption is an explicit parameter in the protocol specification. As is the case with many such protocols, the ICC protocols are easily modified so to adaptively adjust to an unknown communication-delay bound. However, some care must be taken in this, and we discuss this matter in some detail.

We also analyze the *message complexity* of each of the ICC protocols. Message complexity is defined to be the total number of messages sent by all honest parties in any one round — so one party broadcasting a message contributes a term of n to the message complexity. In the worst case, the message complexity is $O(n^3)$. However, we show that in any round where the network is synchronous, the *expected* message complexity is $O(n^2)$ — in fact, it is $O(n^2)$ with overwhelming probability. The probability here is taken with respect to the random beacon for that round.

Of course, message complexity by itself does not tell the whole story of communication complexity: the sizes of the messages is important, as is the communication pattern. In each of the protocols ICC0 and ICC1, in any one round, each honest party broadcasts $O(n)$ messages in the worst case, where each message is either a signature, a signature share (for a threshold or multi-signature scheme), or a block. Signatures and signature shares are typically very small (a few dozen bytes) while blocks may be very large (a block's payload may typically be a few megabytes). If the network is synchronous in that round, each honest party broadcasts $O(1)$ such messages (both small and large) with overwhelming probability. Moreover, the total number of *distinct* blocks broadcast by all the honest parties is typically $O(1)$ — that is, the honest parties typically all broadcast the same block (or one of a small handful of distinct blocks). This property interacts well with the Internet Computer's implementation of these broadcasts, which is done using a peer-to-peer gossip sub-layer [DGH$^+$87]. As we will discuss, Protocol ICC1 is explicitly designed to coordinate well with this peer-to-peer gossip sub-layer (even though the logic of the protocol can be easily understood independent of this sub-layer).

Protocol ICC2 has very much the same structure as Protocol ICC1; however, instead of relying on a peer-to-peer gossip sub-layer to efficiently disseminate large blocks, it instead makes use of subprotocol based on *erasure codes* to do so. Assuming blocks have size S, and that $S = \Omega(n \log n\, \lambda)$, where signatures (and hashes) have length $O(\lambda)$, the total number of bits transmitted by each party in each round of ICC2 is $O(S)$ with overwhelming probability (assuming the network is synchronous in that round).

We also analyze the *reciprocal throughput* and *latency* of the ICC protocols. In a steady state of the system where the leader is honest and the network delay is bounded by $\delta \leq \Delta_{\text{bnd}}$, Protocols ICC0 and ICC1 will finish a round once every 2δ units of time. That is, the reciprocal throughput is 2δ. The latency for these protocols, that is, the elapsed time from when a leader proposes a block and when all parties commit to a block, is 3δ. For Protocol ICC2, the reciprocal throughput is 3δ and the latency is 4δ. The bound δ may be much smaller than the network-delay bound Δ_{bnd} on which the partial synchrony assumption (used to ensure liveness) is based. In particular, the ICC protocols enjoy the property known as *optimistic responsiveness* [PS18], meaning that the protocol will run *as fast as the network will allow* in those rounds where the leader is honest. For an arbitrary round, where the leader is not honest or $\delta > \Delta_{\text{bnd}}$, the round will finish in time $O(\Delta_{\text{bnd}} + \delta)$ with overwhelming probability.

1.1 Related work

The atomic broadcast problem is a special case of what is known as the *consensus* problem. Reaching consensus in face of arbitrary failures was formulated as the Byzantine Generals Problem by [LSP82]. The first solution in the synchronous communication model was given by [PSL80].

In the asynchronous communication model, it was shown that no deterministic protocol can solve the consensus problem. Despite this negative result, the problem can be solved by probabilistic protocols. The first such protocol was given by [Ben83], who also showed that the resilience bound $t < n/3$ is optimal in the asynchronous setting. More efficient protocols can be achieved using cryptography, as was shown in [CKS05, CKPS01], with significant improvements more recently in [MXC$^+$16, DRZ18, GLT$^+$20].

Despite the recent progress made in the asynchronous setting, much more efficient consensus protocols are available in the partially synchronous setting. The goal in this setting is to guarantee safety without making any synchrony assumptions, and to rely on periods of network synchrony only to guarantee liveness. The first consensus protocol in the partially synchronous setting was given by [DLS88]. The first truly practical protocol in this setting is the well-known PBFT protocol [CL99, CL02], which is a protocol for atomic broadcast and state machine replication.

PBFT proceeds in rounds. In each round, a designated leader proposes a batch of commands by broadcasting the batch to all parties. This is followed by two all-to-all communication steps to actually commit to the batch. Under normal operation, the leader will continue in its role for many rounds. However, if sufficiently many parties determine that the protocol is not making timely progress, they will trigger a *view-change* operation, which will install a new leader, and clean up any mess left by the old leader.

Despite its profound impact on the field, there are several aspects where PBFT leaves some room for improvement.

1. The view-change subprotocol is a relatively complicated and expensive procedure.
2. The leader is responsible for disseminating the batch to all parties. This creates two problems.
   (a) First, if the batches are very large, the leader becomes the bottleneck in terms of communication complexity.
   (b) Second, a corrupt leader can fail to disseminate a batch to all parties. In fact, a corrupt leader (together with the help of other corrupt parties) can easily drive the protocol forward an arbitrary number of rounds, and leave a subset of the honest parties lagging behind without any of the batches corresponding to those rounds. The details of how these lagging parties catch up are not specified, other than to say that such a party can obtain any missing batch from another. While this is certainly true, a naive implementation of this idea makes it easy for an attacker to drive up the communication complexity even further, by making many corrupt parties request missing batches from many honest parties — so instead of just the leader broadcasting the batches, one could end up in a setting where $O(n)$ honest parties are each transmitting a batch to $O(n)$ corrupt parties in every round.

3. The all-to-all communication pattern in the last two steps of each round can also result in high communication complexity. However, this need not be the case if the batches are very large relative to n — in this case, the dissemination of the batches is still the dominant factor in terms of communication complexity.

The *communication complexity* of a protocol is traditionally defined as the total number of bits transmitted by all honest parties. In a protocol such as PBFT, which is structured in rounds, this is typically measured on a per-round basis.

A lot of work has gone into reducing the communication complexity of PBFT by eliminating the all-to-all communication steps [RC05, GAG$^+$19, AMN$^+$20]. However, as was demonstrated empirically in [SDV19], this effort may be misplaced: in terms of improving throughput and latency, it is not the communication complexity that is important, but rather, the communication *bottlenecks*. That is, the relevant measure is not the *total* number of bits transmitted by all parties, but the *maximum* number of bits transmitted *by any one party*. Such empirical findings are of course sensitive to the characteristics of the network. In [SDV19], the network was a global wide-area network, which is the setting of most interest to us in this work. As was reported in [SDV19], it is the dissemination of large batches that creates a communication bottleneck at the leader, and not the all-to-all communication steps, which involve only smaller objects. In fact, [SDV19] argue that approaches such as those in [RC05, GAG$^+$19, AMN$^+$20] only exacerbate the bottleneck at the leader.

There has also been recent work on eliminating the complex view-change subprotocol of PBFT, for example HotStuff [AMN$^+$20] and Tendermint [BKM18]. Like PBFT, both of these protocols are leader based; however, they do not rely on a complicated and expensive view-change subprotocol, and in fact may rotate through a new leader every round. Unlike PBFT, both of these protocols are blockchain based protocols (while PBFT can be used in the context of blockchains, it need not be).

HotStuff eliminates the all-to-all communication steps of PBFT. Also, HotStuff (actually, "chained" HotStuff, which is a pipelined version of HotStuff) improves on the throughput of PBFT, reducing the reciprocal throughput from 3δ to 2δ, where δ is the network delay. Like PBFT, HotStuff is optimistically responsive (it runs as fast as the network will allow when the leader is honest). Note, however, that the latency (the elapsed time between when a leader proposes a block and when it is committed) of HotStuff increases from 3δ to 6δ. Like PBFT, HotStuff relies on the leader to disseminated blocks (i.e., batches), and just as for PBFT, this can become a communication bottleneck, and there is no explicit mechanism to ensure blocks are reliably disseminated when the leader is corrupt. In addition, while HotStuff does not rely on a "view change" subprotocol, it still relies on something called a "pacemaker" subprotocol. While the task of the pacemaker subprotocol is less onerous than that of the view-change subprotocol, it is still decidedly nontrivial, and is not specified in [AMN$^+$20]. LibraBFT (a.k.a., DiemBFT) [Lib20] implements a pacemaker subprotocol, but that subprotocol re-introduces the very all-to-all communication pattern that HotStuff intended to eliminate. More recently, pacemaker protocols have been proposed with better communication complexity [NBMS19, NK20, BCG20]. Note that none of these proposals deal with the reliable and efficient dissemination of blocks or batches, only the synchronization of parties as they move from one round to the next.

Tendermint relies on a peer-to-peer gossip sub-layer for communication. One advantage of this is that the reliable dissemination of blocks proposed by a leader is built into the protocol, unlike protocols such as PBFT and HotStuff. Moreover, a well-designed gossip sub-layer can significantly reduce the communication bottleneck at the leader — of course, this may come at the cost of increased reciprocal throughput and latency, as dissemination of a message through a gossip sub-layer can take several hops through the underlying physical network. Also, unlike HotStuff, Tendermint does not rely on any unspecified components (like the "pacemaker" in HotStuff). One disadvantage of Tendermint is that unlike PBFT and HotStuff, it is not optimistically responsive. This can be a problem, since to guarantee liveness, one generally has to choose a network-delay upper bound $\Delta_{\rm bnd}$ that may be significantly larger than the actual network delay δ, and in Tendermint, every round takes time $O(\Delta_{\rm bnd})$, even when the leader is honest.

MirBFT [SDV19] is an interesting variant of PBFT in which many instances of PBFT are run concurrently. The motivation for this is to alleviate the bottleneck observed at the leader in ordinary PBFT. Since MirBFT relies on PBFT, it also uses the same complex and expensive view-change subprotocol — however, as pointed out in [SDV19], other protocols besides PBFT could be used in their framework. Having many parties propose batches simultaneously presents new challenges, one of which is to prevent duplication of commands, which can negate any improvements in throughput. A solution to this problem is given in [SDV19].

Algorand [GHM+17] is a system for proof-of-stake blockchain consensus with a number of varied goals, but at its core is a protocol for atomic broadcast. Like Tendermint, it is based on a gossip sub-layer and dissemination of blocks is built into the protocol. Also like Tendermint, it is *not* optimistically responsive. Unlike all of the other protocols discussed here, it relies on a (very weak) synchrony assumption to guarantee safety. Like the ICC protocols, Algorand also uses something akin to a random beacon to rank parties, but the basic logic of how these rankings are used is quite different.

We now highlight the main features of the ICC family of protocols, and how they relate to some of the protocols discussed above.

- The ICC protocols are extremely simple and entirely self contained. There are no complicated subprotocols (similar to view-change in PBFT) and no unspecified subprotocols (such as pacemaker in HotStuff, or reliable batch/block dissemination, as in PBFT and HotStuff).

- As just mentioned, the ICC protocols explicitly deal with the block dissemination problem. Like Tendermint and Algorand, Protocol ICC1 is designed to be integrated with a peer-to-peer gossip sub-layer. As discussed above, such a gossip sub-layer can reduce the communication bottleneck at the leader.

  Instead of a gossip sub-layer, Protocol ICC2 relies on a subprotocol for reliable broadcast that uses erasure codes to reduce both the overall communication complexity and the communication bottleneck at the leader. Such reliable broadcast protocols were introduced in [CT05], and previously used in the context of atomic broadcast in [MXC+16]. We propose a new erasure-coded reliable broadcast subprotocol with better latency than that in [CT05], and with stronger properties that we exploit in its integration with Protocol ICC2.

- Like PBFT and HotStuff, and unlike Tendermint and Algorand, all of the ICC protocols are optimistically responsive. Protocols ICC0 and ICC1 attain a reciprocal throughput of 2δ and a latency of 3δ (when the leader is honest and the network is synchronous). For Protocol ICC2, the numbers increase to 3δ and 4δ, respectively.
- Like PBFT, but unlike HotStuff, the ICC protocols utilize an all-to-all transmission of signatures and signature shares. However, the ICC protocols are geared toward a setting where the blocks are quite large, and so the contribution to the communication complexity of the all-to-all transmissions are typically not a bottleneck. Rather, the communication bottleneck is the dissemination of the blocks themselves, which Protocol ICC1 mitigates by using a gossip sub-layer, while Protocol ICC2 mitigates by using an erasure-coded reliable broadcast subprotocol.
- Unlike all of the protocols discussed above, for the ICC protocols, in every round, at least one block is added to a block-tree, and one of these blocks will eventually become part of the chain of committed blocks. This ensures that the overall throughput remains fairly steady, even in periods of asynchrony or in rounds where the leader is corrupt. That said, in a round with a corrupt leader, the block proposed by the leader may not be as useful as it would be if the leader were honest; for example, at one extreme, a corrupt leader could always propose an empty block. However, if a leader consistently underperforms in this regard, the Internet Computer provides mechanisms for reconfiguring the set of protocol participants (which are not discussed here), by which such a leader can be removed.

Robust consensus. We note that the simple design of the ICC protocols also ensures that they degrade quite gracefully when and if Byzantine failures actually do occur. As pointed out in [CWA$^+$09], much of the recent work on consensus has focused so much on improving the performance in the "optimistic case" where there are no failures, that the resulting protocols are dangerously fragile, and may become practically unusable when failures do occur. For example, [CWA$^+$09] show that the throughput of existing implementations of PBFT drops to zero under certain types of (quite simple) Byzantine behavior. The paper [CWA$^+$09] advocates for *robust* consensus, in which *peak* performance under optimal conditions is partially sacrificed in order to ensure *reasonable* performance when some parties actually are corrupt (but still assuming the network is synchronous). The ICC protocols are indeed robust in the sense of [CWA$^+$09]: in any round where the leader is corrupt (which itself happens with probability less than 1/3), each ICC protocol will effectively allow another party to take over as leader for that round, with very little fuss, to move the protocol forward to the next round in a timely fashion. The only performance degradation in this case is that instead of finishing the round in time $O(\delta)$, where δ is the actual network delay, the round will finish (with overwhelming probability) in time $O(\Delta_{\rm bnd})$, where $\Delta_{\rm bnd} \geq \delta$ is the network-delay bound on which the partial synchrony assumption (used to ensure liveness) is based.

Preliminary versions of the ICC protocols. Note that the protocols presented here are very different from those discussed in either [HMW18] or [AMNR18]. In particular, unlike the protocols presented here, the preliminary protocols in [HMW18, AMNR18] (1) only guaranteed safety in a synchronous setting, (2) were not optimistically responsive, and (3) had potentially unbounded communication complexity.

1.2 Outline of the remainder of the paper

In Section 2, we review the cryptographic primitives we will need for our protocols. In Section 3, we present Protocol ICC0. In Section 4, we give a detailed analysis of Protocol ICC0. Section 5 presents several variations of ICC0. Two major variations, Protocols ICC1 and ICC2, are discussed in detail in Sections 5.2 and 5.3.

2 Cryptographic primitives

2.1 Collision resistant hash function

Our protocols use a hash function H that is assumed to be *collision resistant*, meaning that it is infeasible to find two distinct inputs that hash to the same value; i.e., it is infeasible to find inputs x, x' with $x \neq x'$ but $H(x) = H(x')$.

2.2 Digital signatures

Our protocols use a digital signature scheme that is secure in the standard sense that it is infeasible to create an existential forgery in an adaptive chosen message attack.

2.3 Threshold signatures

A (t, h, n)-**threshold signature scheme** is a scheme in which n parties are initialized with a public-key/secret-key pair, along with the public keys for all n parties, as well as a global public key.

- There is a **signing algorithm** that, given the secret key of a party and a message m, generates a **signature share on** m.
- There is also a **signature share verification algorithm** that, given the public key of a party, along with a message m and a signature share ss, determines whether or not ss is a **valid signature share on** m under the given public key.

  It is required that correctly generated signature share are always valid.
- There is a **signature share combining algorithm** that, given valid signature shares from h different parties on a given message m, combines these signature shares to form a **signature on** m.
- There is a **signature verification algorithm** that, given the global verification key, along with a signature σ and a message m, determines if σ is a **valid signature on** m.

  It is required that if the combining algorithm combines valid signature shares from h distinct parties, then (with overwhelming probability) the result is a *valid* signature on m.

We say that such a scheme is **secure** if it is infeasible for an efficient adversary to win the following game.

- The adversary begins by choosing a subset of t "corrupt" parties. Let us call the remaining $n - t$ parties "honest".
- The challenger then generates all of the key material, giving the adversary all of the public keys, as well as the secret keys for the corrupt parties.
- The adversary makes a series of signing queries. In each such query, the adversary specifies a message and an honest party. The challenger responds with a signature share for that party on the specified message.
- At the end of the game, the adversary outputs a message m and a signature σ.
- We say that the adversary *wins the game* if σ is a valid signature on m, but the adversary obtained signature shares on m from fewer than $h - t$ honest parties.

Threshold signatures of this type can be implemented in several ways.

(i) One way is simply to use an ordinary signature scheme to generate individual signature shares, and the combination algorithm just outputs a set of signature shares.

(ii) A second way is to use multi-signatures, such as BLS multi-signatures [BDN18], in which a signature share is an ordinary BLS signature [BLS01], which can be combined into a new BLS signature on an aggregate of the individual public keys, together with a descriptor of the h individual signatories.

(iii) A third approach is to use an ordinary signature scheme such as BLS, but with the secret key shared (via Shamir secret sharing [Sha79]) among the parties.

There are various trade-offs among these approaches:

- Unlike (iii), approaches (i) and (ii) have the advantage of not requiring any trusted setup or distributed key generation protocol.
- Unlike (iii), signatures in approaches (i) and (ii) identify the signatories (which can be either a "bug" or a "feature").
- Signatures of type (iii) are unique (if the signatures of the underlying non-threshold scheme is unique, which is the case for BLS signatures). Signatures of type (i) and (ii) are not unique.
- The signatures in (iii) are typically (e.g., for BLS) more compact than those in (i) or (ii).
- Finally, for $h > t + 1$, the security of approach (iii) may depend on somewhat stronger (though still reasonable) security assumptions.

For the security of our atomic broadcast protocols, we use both approaches (ii) and (iii).

We use approach (ii) with $h = n - t$ for authorization purposes: when a party wishes to authorize a given message, it broadcasts a signature share on a message. Assuming the scheme is secure, the existence of a valid signature on a message means that at least $n - 2t$ honest parties must have authorized the message.

We use approach (iii) to build a **random beacon**. For this, we need unique signatures (which BLS provides). A random beacon is a sequence of values $R_0, R_1, R_2 \ldots$. The value R_0 is a fixed, initial value, known to all parties. For $k = 1, 2, \ldots,$ the value R_k is the threshold signature on R_{k-1}. When a party has R_{k-1} and wishes to generate R_k, it broadcasts its signature share on the message R_{k-1}. If $t + 1$ honest parties in total do the same, they can each construct the value R_k. However, assuming the threshold signature scheme is secure, unless at least one honest party contributes a signature share, the value R_k cannot be constructed, and in fact, a hash of R_k will be indistinguishable from a random string (if we model the hash function as a "random oracle" [BR93]).

3 Protocol ICC0

In this section, we present our Protocol ICC0 for atomic broadcast in detail.

3.1 Preliminaries

Interval notation. Throughout this paper, we use the notation $[k]$ to denote the set $\{0, \ldots, k - 1\}$.

We have n parties, $P_1, \ldots, P_n$. It is assumed that there are at most t corrupt parties. We shall assume a *static* corruption model, where an *adversary* decides at the outset of the protocol execution which parties to corrupt. We shall generally assume *Byzantine failures*, where a corrupt party may behave arbitrarily, and where all the corrupt parties are coordinated by the adversary. However, we shall sometimes consider weaker forms of corruption, such as *crash failures*, in which corrupt parties are simply non-responsive. We shall also have occasion to consider an intermediate form of corruption called *consistent failures*, which is somewhat protocol specific, but generally means that a corrupt party behaves in a way that is not conspicuously incorrect (see Section 4.7).

The only type of communication performed by our protocol is *broadcast*, wherein a party sends the same message to all parties (this will apply to both Protocols ICC0 and ICC1, but not ICC2). This is not a secure broadcast: if the sender is corrupt, there are no guarantees that the honest parties will receive the same message or any message at all; if the sender is honest, then all honest parties will eventually receive the message. We shall generally assume that the precise scheduling of message delivery is determined by the adversary.

Each party has a *pool* which holds the set of all messages received from all parties (including itself). As we describe our protocol, no messages are ever deleted from a pool. While the protocol can be optimized so that messages that are no longer relevant may discarded, we do not discuss those details here. In addition, a practical implementation of a replicated state machine would typically incorporate some kind of checkpointing and garbage collection mechanism, similar to that in PBFT [CL99]. Again, we do not discuss these details. Although the Internet Computer implementation uses a “gossip network” to transmit messages among parties, we shall not make any assumptions about the underlying network, except those already mentioned above.

Each party will be initialized with some secret keys, as well as with the public keys for itself and all other parties. For some cryptographic primitives, the secret keys of the parties are correlated with one another, and must either be set up by a trusted party or a secure distributed key generation protocol.

Some of these cryptographic keys are for digital signatures, which are used to authenticate messages. No other message authentication mechanism is required.

3.2 Components

Our protocol uses:

- a collision resistant hash function H;
- a signature scheme S_{auth}, where each honest party has a secret key, and is provisioned with the public keys of all parties;
- an instance S_{notary} of a $(t, n-t, n)$-threshold signature scheme, where each honest party has a secret key, and is provisioned with all of the public key material for the instance;
- an instance S_{final} of a $(t, n-t, n)$-threshold signature scheme, where each honest party has a secret key, and is provisioned with all of the public key material for the instance;
- an instance S_{beacon} of a $(t, t+1, n)$-threshold signature scheme, where each honest party has a secret key, and is provisioned with all of the public key material for the instance; this is used to implement a *random beacon*, as described in Section 2; as such, the scheme is required to provide unique signatures.

3.3 High level description of the protocol

The protocol proceeds in rounds. In each round, each party may propose a *block* to be added to a *block-tree*. Here, a block-tree is a directed rooted tree. Except for the root, each node in the tree is a block, which consists of

- a round number (which is also the depth of the block in the tree),
- the index of the party who proposed the block,
- the hash of the block's parent in the block-tree (using the collision resistant hash function H),
- the *payload* of the block.

The root itself is a special block, denoted root.

The details of the payload of a block are application dependent. In the context of atomic broadcast, as described in Section 1, the payload would naturally consist of one or more commands that have been input to the party proposing the block. Moreover, in constructing the payload for a proposed block, a party is always extending a particular path in the block-tree, and can take into account the payloads in the blocks already in that path (for example, to avoid duplicating commands). This is an important feature for state machine replication.

To propose a block, a party must sign the block with a digital signature (using S_{auth}). To add a proposed block to the block-tree, the block must be *notarized* by a quorum of $n - t$ parties, using the threshold signature scheme S_{notary}. Further, a notarized block may be *finalized* by a quorum of $n - t$ parties, using the threshold signature scheme S_{final}.

A random beacon is also used, implemented using the threshold signature scheme S_{beacon}, so that in each round, the next value of the random beacon is revealed. The value of the random beacon in a given round determines a permutation π on the parties, which assigns a unique rank $0, \ldots, n-1$ to each party. Under cryptographic assumptions, the permutation π in each round is effectively a random permutation, and independent of the permutations used in previous rounds, and independent of the choice of corrupt parties (this assumes an adversary that *statically* corrupts parties).

The party of rank 0 is the *leader* for that round. While the protocol gives priority to a block proposed by the leader of the round, other parties may propose blocks as well. In particular, if the leader is corrupt or temporarily cut off from the network, blocks proposed by other parties will be notarized and possibly finalized.

As we will see, under certain cryptographic assumptions, but without any synchrony assumptions, it is guaranteed that in each round $k \geq 1$:

P1: at least one notarized block of depth k will be added to the block-tree, and

P2: if a notarized block of depth k is finalized, then there is no other notarized block of depth k.

Moreover, we have:

P3: if the network is synchronous over a short interval of time beginning at the point in time where any honest party first enters round k, and the leader in round k is honest, then the block proposed by the leader in round k will be finalized.

Property P1 ensures that the protocol does not deadlock, in that the tree grows in every round.

Property P2 is used as follows. Suppose some party sees a finalized depth-k block B, and let p the path in the block-tree from the root to B. Suppose some party (either the same or a different one) sees a finalized depth-k' block B', where $k' \geq k$, and let p' the path in the block-tree from the root to B'. Then Property P2 implies that the path p must be a prefix of path p': if it were not, then there would be two distinct notarized blocks at depth k, contradicting Property P2.

In the context of atomic broadcast, the above argument shows that when a party sees a finalized block B, it may safely append to its output queue the commands in the payloads of the blocks on the path leading from the root to B, in that order.

Property P3 guarantees a strong notion of liveness under a partial synchrony assumption. Indeed, if at least $n - t$ parties have received a command as input by round k, then at least $n - 2t > n/3$ honest parties will have received that command as input, and so with probability $> 1/3$, the leader for round k can ensure that this command is in its proposed block, and if the synchrony assumption holds for round k, each honest party will output this command in round k (as soon as all relevant messages have been delivered).[1]

Moreover, because of Property P1, even if the network remains asynchronous for many rounds, as soon as it becomes synchronous for even a short period of time, the commands from the payloads of all of the rounds between synchronous intervals will be output by all honest parties. Thus, even if the network is only intermittently synchronous, the system will maintain a constant throughput. However, to the extent that the blocks in the rounds in-between are proposed only by corrupt parties, the commands from these rounds may not be of much use.

3.4 Blocks

We now give more details on blocks. There is a special round-0 block root.

For $k \geq 1$, a round-k block B is a tuple of the form

$$(\mathsf{block}, k, \alpha, \mathit{phash}, \mathit{payload}). \tag{1}$$

Here, α represents the index of the party P_α who proposed this block, *phash* is an output of the hash function H, and *payload* is application-specific content.

We classify a block in an honest party Q's pool as *authentic*, *valid*, *notarized*, or *finalized* (for Q), depending on other data in Q's pool. The special root is always present in Q's pool, and is always considered authentic, valid, notarized, and finalized (for Q).

Let $k \geq 1$ and let B be a round-k block B as in (1) in Q's pool.

- B is called **authentic (for Q)** if there is an *authenticator for B* in Q's pool.

  An **authenticator for** B is a tuple $(\mathsf{authenticator}, k, \alpha, H(B), \sigma)$, where σ is a valid S_{auth}-signature on $(\mathsf{authenticator}, k, \alpha, H(B))$ by party P_α.

- B is called **valid (for Q)** if it is *authentic (for Q)*, and if $\mathit{phash} = H(B_\text{p})$ for some round-$(k-1)$ block B_p in Q's pool that is *notarized (for Q)*. B_p is called the **parent** of B and we say B **extends** B_p.

  Note that by the collision resistance property of H, we may assume that B's parent is unique.

  Also note that there may be an application-specific property that must be satisfied in order to consider B to be valid.

- B is called **notarized (for Q)** if it is *valid* and there is a *notarization for B* in Q's pool.

  A **notarization for** B is a tuple $(\mathsf{notarization}, k, \alpha, H(B), \sigma)$, where σ is a valid S_{notary}-signature on $(\mathsf{notarization}, k, \alpha, H(B))$. A **notarization share for** B is a tuple $(\mathsf{notarization\text{-}share}, k, \alpha, H(B), \mathit{ns}, \beta)$, where *ns* is a valid S_{notary}-signature share on $(\mathsf{notarization}, k, \alpha, H(B))$ by party P_β.

[1]This presumes that there is no limit on the size of a payload. If there is a limit, but honest parties give priority to older commands, a reasonably strong notion of liveness will still be satisfied.

- B is called **finalized (for Q)** if it is *valid (for Q)* and there is a *finalization for B* in Q's pool.

  A **finalization for** B is a tuple $(\mathsf{finalization}, k, \alpha, H(B), \sigma)$, where σ is a valid S_{final}-signature on $(\mathsf{finalization}, k, \alpha, H(B))$. A **finalization share for** B is a tuple $(\mathsf{finalization\text{-}share}, k, \alpha, H(B), \mathit{fs}, \beta)$, where fs is a valid S_{final}-signature share on $(\mathsf{finalization}, k, \alpha, H(B))$ by party P_β.

In what follows, root serves as its own authenticator, notarization, and finalization.

Notice that if a party has a valid round-k block B in its pool, then there are also blocks $\mathsf{root} = B_0, B_1, \dots, B_k = B$ in its pool that form a **blockchain**, meaning that B_i is B_{i+1}'s parent, for $i = 0, \dots, k-1$, along with authenticators for $B_1, \dots, B_k$ and notarizations for $B_1, \dots, B_{k-1}$.

3.5 Protocol details

The protocol consists of two subprotocols that run concurrently: the *Tree Building Subprotocol* and the *Finalization Subprotocol*.

The Tree Building Subprotocol for party P_α is shown in Figure 1. The Tree Building Subprotocol makes use of two *delay functions*:

- $\Delta_{\text{prop}} : [n] \to \mathbb{R}_{\geq 0}$ is used to delay proposing a block, based on the rank of the proposer. It should be a non-decreasing function.

- $\Delta_{\text{ntry}} : [n] \to \mathbb{R}_{\geq 0}$ is used to delay generating a notarization share on a block, based on the rank of the proposer. It should be a non-decreasing function.

Our presentation and analysis of our protocol will be in terms of these general delay functions. Looking ahead, for liveness, the only requirement is that $2\delta + \Delta_{\text{prop}}(0) \leq \Delta_{\text{ntry}}(1)$, where δ is a bound on the network delay during that round. However, to better control the communication complexity of the protocol, a recommended implementation of these functions is as follows:

$$\begin{aligned} \Delta_{\text{prop}}(r) &\coloneqq 2\Delta_{\text{bnd}} r; \\ \Delta_{\text{ntry}}(r) &\coloneqq 2\Delta_{\text{bnd}} r + \epsilon. \end{aligned} \tag{2}$$

The above liveness requirement will be satisfied for those rounds where the network delay is bounded by $\delta \leq \Delta_{\text{bnd}}$. The parameter ϵ is a "governor" — it can be set to zero, but setting it to a non-zero value will keep the protocol from running "too fast".

We remind the reader that the only type of communication performed by our protocol is broadcast, wherein a party sends the same message to all parties. Moreover, this broadcast is not assumed to be secure: a party receiving a message from a corrupt party cannot be sure that other parties will receive the same message.

In this protocol description, a party waits for its pool to contain messages satisfying certain conditions. As already discussed, this pool holds the set of all messages received from any party (including messages broadcast by itself), and no messages are ever deleted from a pool (although a properly optimized version of the protocol would do so).

```
broadcast a share of the round-1 random beacon
For each round $k = 1, 2, 3 \ldots$ :
    wait for $t + 1$ shares of the round-$k$ random beacon
    compute the round-$k$ random beacon (which defines the permutation $\pi$ for round $k$)
    broadcast a share of the random beacon for round $k + 1$

    let $r_{\mathrm{me}}$ be the rank of $P_\alpha$ according to the permutation $\pi$
    $\mathcal{N} \leftarrow \emptyset$   // the set of blocks for which notarization shares have been broadcast by $P_\alpha$
    $\mathcal{D} \leftarrow \emptyset$   // the set of ranks disqualified by $P_\alpha$

    $done \leftarrow \mathsf{false}$
    $proposed \leftarrow \mathsf{false}$
    $t_0 \leftarrow$ clock()

    repeat
        wait for either:
            (a) a notarized round-$k$ block $B$,
                  or a full set of notarization shares for some valid
                  but non-notarized round-$k$ block $B$:
                // Finish the round
                combine the notarization shares into a notarization for $B$, if necessary
                broadcast the notarization for $B$
                $done \leftarrow \mathsf{true}$
                if $\mathcal{N} \subseteq \{B\}$ then broadcast a finalization share for $B$
            (b) not $proposed$ and clock() $\geq t_0 + \Delta_{\mathrm{prop}}(r_{\mathrm{me}})$:
                // Propose a block
                choose a notarized round-$(k-1)$ block $B_{\mathrm{p}}$
                $payload \leftarrow getPayload(B_{\mathrm{p}})$
                create a new round-$k$ block $B = (\mathsf{block}, k, \alpha, H(B_{\mathrm{p}}), payload)$
                broadcast $B$, $B$'s authenticator, and the notarization for $B$'s parent
                $proposed \leftarrow \mathsf{true}$
            (c) a valid round-$k$ block $B$ of rank $r$ such that
                  $B \notin \mathcal{N}$, $r \notin \mathcal{D}$, clock() $\geq t_0 + \Delta_{\mathrm{ntry}}(r)$, and
                  there is no valid round-$k$ block $B^*$ of rank $r^* \in [r] \setminus \mathcal{D}$:
                // Echo block $B$
                //    and either broadcast a notarization share for it or disqualify its rank
                if $r \neq r_{\mathrm{me}}$ then
                    broadcast $B$, $B$'s authenticator, and the notarization for $B$'s parent
                if some block in $\mathcal{N}$ has rank $r$
                    then   $\mathcal{D} \leftarrow \mathcal{D} \cup \{r\}$
                    else   $\mathcal{N} \leftarrow \mathcal{N} \cup \{B\}$, broadcast a notarization share for $B$
    until $done$
```

Figure 1: ICC0: Tree Building Subprotocol for party P_α

In each round of the Tree Building Subprotocol, as a preliminary step, party P_α will begin by waiting for $t + 1$ shares of the threshold signature used to compute the random beacon for that round. After that, it will compute the random beacon for round k, and immediately broadcast its share of the random beacon for round $k + 1$. This is a bit of "pipelining" logic used to minimize the latency — as a result of this, the adversary may already know the random beacon for round $k + 1$ well before any honest party has finished round k, but this is not an issue (at least, assuming static corruptions).

As already discussed, the random beacon for round k determines a permutation π of the parties, which assigns a unique **rank** $0, \ldots, n - 1$ to each party. The party of rank 0 is called the **leader** for round k. For a round-k block B, we define $\mathrm{rank}_\pi(B)$ to be the rank of party who proposed B.

Now round k begins in earnest. For this round, party P_α will maintain a set $\mathcal{N}$ of blocks for which it has already broadcast notarization shares, and a set $\mathcal{D}$ of disqualified ranks. If a rank is disqualified, it means that the party of that rank has been caught proposing two different blocks for this round.

The round will end for party P_α as soon as it finds either a notarized round-k block B in its pool, or a full set of $n - t$ notarization shares for some valid but non-notarized round-k block B in its pool. In the latter case, party P_α will combine the notarization shares into a notarization on B, and in either case, it will broadcast the notarization on B. In addition, if party P_α has not itself broadcast a notarization share on any block besides B, it will broadcast a finalization share on B.

Party P_α will propose its own block when $\Delta_{\mathrm{prop}}(r_{\mathrm{me}})$ time units have elapsed since the beginning of the round (more precisely, since the time at which it executes the step $t_0 \leftarrow \mathrm{clock}()$ in Figure 1). This delay is not essential for safety or liveness, but is intended to prevent all honest parties from flooding the network with their own proposals. In particular, when the leader is honest, the delay functions are chosen appropriately, and the network is synchronous, no party other than the leader will broadcast its own block. In proposing its own block, P_α must first choose a notarized round-$(k-1)$ block in B_{p} in its pool to extend. There will always be such a block, since the previous round ends only when there is such a block (or $k = 1$ and $B_{\mathrm{p}} = \mathsf{root}$). There may be more than one such notarized block, in which case it does not matter which one is chosen. Next, P_α must compute a payload. In Figure 1, this is done by calling the function $\mathit{getPayload}(B_{\mathrm{p}})$, the details of which are application dependent, but note that it may depend on B_{p} and the entire chain of blocks ending at B_{p} (for example, to avoid duplicate commands). Finally, having constructed a block B to propose, party P_α broadcasts B, B's authenticator, and the notarization for B's parent B_{p}.

Finally, party P_α will **echo** a valid round-k block B of rank r in its pool provided (i) it has not already broadcast a notarization share for B, (ii) it has not disqualified the rank r, (iii) at least $\Delta_{\mathrm{ntry}}(t)$ time units have passed since the beginning of the round, and (iv) there is no "better" block in it pool. Here, a "better" block would be a valid round-k block whose rank is less than r but has not yet been disqualified. If these condition holds, party P_α does the following:

- it "echoes" the block B, meaning that it broadcasts B, B's authenticator, and the notarization for B's parent;

```
$k_{\max} \leftarrow 0$ // max round finalized by $P_\alpha$
repeat
    wait for:
      (i)  a finalized round-$k$ block $B$ with $k > k_{\max}$, or
      (ii) a complete set of finalization shares for some valid but non-finalized
           round-$k$ block $B$ with $k > k_{\max}$:
        // Commit to the last $k - k_{\max}$ blocks in the chain ending at $B$
        combine the finalization shares into a finalization for $B$, if necessary
        broadcast the finalization for $B$
        output the payloads of the last $k - k_{\max}$ blocks in the chain ending at $B$
        $k_{\max} \leftarrow k$
forever
```

Figure 2: Finalization Subprotocol for party P_α

- in addition, it broadcasts a notarization share for B, unless it has already broadcast a notarization share for a different block of the same rank r, in which case it disqualifies the rank r.

Note that P_α will echo B even if it has already broadcast a notarization share of another block of the same rank. This is to ensure that all other honest parties get a chance to also disqualify rank r. However, note that P_α will echo at most 2 blocks of any given rank.

The Finalization Subprotocol for party P_α is shown in Figure 2. Party P_α tracks the last round $k_{\max}$ for which it has seen a finalized block. Whenever it sees either (i) a finalized round-k block B in its pool, or (ii) a full set of $n - t$ finalization shares for some valid but non-finalized round-k block B in its pool, where $k > k_{\max}$, it proceeds as follows. In case (ii), it combines the finalization shares into a finalization on B, and in either case (i) or (ii), it will broadcast the finalization on B. In addition, it will output the payloads of the last $k - k_{\max}$ blocks in the blockchain ending at B, in order.

Our formal execution model is that when a "wait for" statement is executed, execution will pause (if necessary) until a message arrives or a timing condition occurs that makes one of the conditions in the "wait for" be satisfied. When that happens, the corresponding clause is executed (if there are several conditions that are satisfied, one is chosen arbitrarily). We will assume that the pool for that party is not modified while this clause is executing.

4 Analysis of Protocol ICC0

4.1 Preliminary results

Lemma 1 (Termination). *In each round, each honest party will execute the main loop $O(n)$ times.*

Proof. When condition (a) in the "wait for" statement in the main loop is triggered, the loop terminates. Condition (b) executes at most once. So consider condition (c). This can trigger at most $2n$ times. To see this, note that each time it triggers, we either

- add a block to $\mathcal{N}$ of rank distinct from any already in $\mathcal{N}$ which can happen at most n times, or
- or we add a new rank to the set $\mathcal{D}$ which can also happen at most n times (and at most t times, assuming at most t corrupt parties and secure signatures, as each such rank belongs to a corrupt party).

□

Lemma 1 gives a worst case bound on the *message complexity per round* of $O(n^3)$. Later, under a partial synchrony assumption, we will prove a bound on the *expected message complexity per round* of $O(n^2)$.

Lemma 2 (Bound on the number of notarizations). *Assume at most $t < n/3$ corrupt parties, secure signatures, and collision resistance. In each round, and for each party, there can be at most one block proposed by that party that is notarized. In particular, in any round, at most n notarized blocks appear anywhere in the system.*

Proof. In each round, an honest party generates a notarization share for at most one block of any given rank. Since the threshold for a notarization is $n - t$ and $t < n/3$, the lemma follows from a standard quorum intersection argument. □

Lemma 3 (Block chains everywhere). *Assume at most $t < n/3$ corrupt parties, secure signatures, and collision resistance. Suppose that at some point in time, some honest party P has broadcast a round-k block B with the blockchain $\mathsf{root} = B_0, B_1, \ldots, B_k = B$. Then we have:*

(i) The blocks $B_1, \ldots, B_k$, along with authenticators for blocks $B_1, \ldots, B_k$ and notarizations for blocks $B_1, \ldots, B_{k-1}$ have already been broadcast by honest parties.

(ii) In particular, when all the blocks, authenticators, and notarizations in (i) have been delivered to any honest party Q, B is a valid block for Q.

Proof. Whenever an honest party broadcasts a block $B = B_k$, it also broadcasts B_k's authenticator and B_{k-1}'s notarization. Moreover, by the fact that this party has B_{k-1}'s notarization, some honest party must have already broadcast a corresponding notarization share for B_{k-1}, and hence must have already broadcast the block B_{k-1}. (i) follows by induction. (ii) follows by definition. □

4.2 Main results

We now prove the main results, corresponding to Properties P1, P2, and P3 discussed in Section 3.3. The following lemma corresponds to Property P1.

Lemma 4 (Deadlock freeness). *Assume at most $t < n/3$ corrupt parties, secure signatures, and collision resistance. In each round, if all messages up to that round broadcast by honest parties have been delivered to all honest parties, and all notarization and proposal delay times have elapsed, then all honest parties will have finished the round with a notarized block.*

Proof. Suppose the theorem is false. Then there is some finite execution which leaves the system in a state where for some round, all messages broadcast by honest parties up through that round have been delivered to all honest parties, and all delays have expired, but there is *some* honest party that has not finished the round with a notarized block. Choose the first such round.

So we know that each honest party actually entered this round. Moreover, since all honest parties have finished the previous round, they all broadcast shares for the random beacon of this round, and hence all honest parties have received sufficiently many shares to reconstruct the random beacon for this round. Hence all honest parties have not only entered this round, but they have entered the main loop of this round.

We also know that no honest party has finished the round with a notarized block, as otherwise, if some honest party did so, it would have broadcast that notarization and all honest parties would have finished the round with a notarization.

Let us name the parties by their rank, so party $P^{(r)}$ is the party of rank r. For any honest party $P^{(r)}$, let us say that a **rank s is good for $P^{(r)}$** if $P^{(r)}$ has exactly one valid block proposed by $P^{(s)}$ for this round in its pool; otherwise, we say **rank s is bad for $P^{(r)}$**. If there is any good rank for $P^{(r)}$, define $s^*(r)$ to be the smallest among them; otherwise, define $s^*(r) := n$.

Claim 1. Let $P^{(r)}$ be an honest party. For each $s < s^*(r)$, either

- $P^{(r)}$ has no valid blocks proposed by $P^{(s)}$ for this round in its pool, or
- $P^{(r)}$ has more than one valid block proposed by $P^{(s)}$ for this round in its pool, $P^{(r)}$ has broadcast two such blocks, and $P^{(r)}$ has disqualified rank s.

Proof. This is clear from the logic of the protocol by induction on s. Recall that we are assuming that $P^{(r)}$ has not finished the round. For each $s < s^*(r)$, if $P^{(r)}$ has more than one valid block proposed by $P^{(s)}$ in its pool, then (by induction) for any rank $t < s$, $P^{(r)}$ would either have no blocks proposed by $P^{(t)}$ in its pool, or would have disqualified rank t, and hence $P^{(r)}$ would have broadcast two blocks proposed by $P^{(s)}$, thus disqualifying rank s.

Claim 2. Let h be the rank of the honest party of least rank. Then $s^*(h) \leq h$.

Proof. We want to show that $s^*(h) < h$ or $s^*(h) = h$, or equivalently, $s^*(h) \geq h$ implies $s^*(h) = h$. So assume $s^*(h) \geq h$. Then by the logic of the protocol and Claim 1, and the assumption that $P^{(h)}$ has not finished the round, $P^{(h)}$ would have proposed and broadcast its own block, which would mean rank h is good for $P^{(h)}$, which means $s^*(h) = h$.

Claim 3. Let $P^{(r)}$ be an honest party and suppose $s := s^*(r) < n$. Then $P^{(r)}$ has a unique valid block proposed by $P^{(s)}$ in its pool, $P^{(r)}$ broadcast that block along with a corresponding notarization share.

Proof. This follows from Claim 1, the logic of the protocol, and the assumption that $P^{(r)}$ has not finished the round.

Claim 4. For any two honest parties $P^{(r)}$ and $P^{(r')}$, we have $s^*(r) = s^*(r')$.

Proof. Let $s := s^*(r)$ and $s' := s^*(r')$. By way of contradiction, suppose, say, $s < s'$. By Claim 3, $P^{(r)}$ has a unique valid block B proposed by $P^{(s)}$ in its pool and $P^{(r)}$ broadcast the block B. Therefore (by Lemma 3), B is also a valid block in the pool of $P^{(r')}$. Since

$s < s'$, we see that rank s is bad for $P^{(r')}$, and therefore, by definition, $P^{(r')}$ must have another valid block B' proposed by $P^{(s)}$ in its pool. Moreover, by Claim 1, $P^{(r')}$ would have broadcast two valid blocks proposed by $P^{(s)}$, and (again, by Lemma 3) both of these would appear as valid blocks in the pool of $P^{(r)}$. This contradicts the assumption that rank s is good for $P^{(r)}$.

Let s^* be the common value of $s^*(r)$ among all honest parties $P^{(r)}$, as guaranteed by the previous claim. By Claim 2, $s^* \leq h$.

Claim 5. There is a block B^* of rank s^* such that for each honest party P, B^* appears as a valid block in P's pool, and at least $n - t$ notarization shares for B^* are also in P's pool.

Proof. By Claim 3, each honest party $P^{(r)}$ has a unique valid block $B^*(r)$ of rank s^* in its pool, and has broadcast $B^*(r)$, along with a corresponding notarization share. Moreover (by Lemma 3), each $B^*(r)$ appears as a valid block in every honest party's pool. Therefore, $B^*(r) = B^*(r')$ for all honest parties $P^{(r)}$ and $P^{(r')}$. Let B^* be the common value of $B^*(r)$. Moreover, each honest party has received at least $n - t$ notarization shares for B^*.

Finally, by Claim 5, we see that each honest party will have finished the protocol. A contradiction. □

The following lemma corresponds to Property P2 discussed in Section 3.3.

Lemma 5 (Safety). *Assume at most $t < n/3$ corrupt parties, secure signatures, and collision resistance. At each round, if some block is finalized, then no other block at that round can be notarized.*

Proof. Suppose $f \leq t < n/3$ parties are corrupt. If a block B is finalized, then at least $n - t - f$ honest parties issued finalization shares for B, which means that these honest parties did not issue notarization shares for any block besides B. If another block were notarized, this would require $n - t - f$ notarization shares from a disjoint set of honest parties, which is impossible, given $f \leq t < n/3$. □

As discussed in Section 3.3, this property implies that the protocol satisfies the safety property.

We next discuss liveness. For this, we formally state our partial synchrony assumption:

Definition 1. *Suppose that at time T, all honest parties have been initiated. We say the communication network is **δ-synchronous at time** T if all messages that have been sent by honest parties by time T arrive at their destinations* before *time $T + \delta$.*

The following lemma corresponds to Property P3 discussed in Section 3.3.

Lemma 6 (Liveness assuming partial synchrony). *Assume that:*

(i) *there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;*

(ii) *$k > 1$, the first honest party P to enter round k does so at time T, and all honest parties have been initiated at time T;*

(iii) *the leader Q of round k is honest;*

(iv) the communication network is δ-synchronous at times T and $T + \delta + \Delta_{\text{prop}}(0)$;

(v) $2\delta + \Delta_{\text{prop}}(0) \leq \Delta_{\text{ntry}}(1)$.

Then when all round-k messages from honest parties have been delivered to all honest parties, each honest party will have Q's round-k proposed block in its pool as a finalized block.

Proof. At time T, P has already broadcast notarizations for blocks at rounds $1, \dots, k-1$. This means that at least $n - 2t > t$ honest parties have already broadcast all these blocks, and hence also shares of random beacons for rounds $1, \dots, k$.

Therefore, before time $T + \delta$, by the synchrony assumption and Lemma 3, for $i = 1, \dots, k-1$, Q has a notarized round-i block in its pool, and for $i = 1, \dots, k$, Q has more than t shares of the round-i random beacon in its pool.

It follows that Q will have entered the main loop of round k before time $T + \delta$, and broadcast its own round-k proposed block B before time $T + \delta + \Delta_{\text{prop}}(0)$.[2] Again, by the synchrony assumption and Lemma 3, B appears as a valid block in every honest party's pool before time $T + 2\delta + \Delta_{\text{prop}}(0) \leq T + \Delta_{\text{ntry}}(1)$. This implies every honest party will broadcast notarization shares for B and only for B.

It follows that whenever all round-k messages from honest parties have been delivered to all honest parties, each honest party will have broadcast finalization shares for B, and hence each honest party will have B in its pool as a finalized block. □

Note that the condition (v) of Lemma 6 will be satisfied by the delay functions defined in (2) when $\delta \leq \Delta_{\text{bnd}}$.

4.3 Expected message complexity and latency

In a given round k, we define the **leading honest party** to be the honest party of lowest rank in that round, and we define $N_{\text{echo}}^{(k)}$ to be the highest rank of any block that is echoed by any honest party in that round. Note that the number of blocks broadcast by any one honest party is $O(N_{\text{echo}}^{(k)})$. The following lemma says that under a partial synchrony assumption, we have $N_{\text{echo}}^{(k)} \leq h$, where h is the rank of the leading honest party in round k. This implies that the message complexity in round k is $O((h+1)n^2)$. Later, we will use this result to prove that the expected message complexity per round is $O(n^2)$ — in fact, it is $O(n^2)$ with overwhelming probability.

Lemma 7 (Message complexity assuming partial synchrony). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) $k > 1$, the first honest party to enter round k does so at time T, and all honest parties have been initiated by time T;

(iii) the leading honest party Q of round k is of rank h;

(iv) the communication network is δ-synchronous at times T and $T + \delta + \Delta_{\text{prop}}(h)$;

[2]Here, we are assuming computation time is negligible relative to network latency.

(v) $2\delta + \Delta_{\text{prop}}(h) \le \Delta_{\text{ntry}}(h+1)$.

Then $N^{(k)}_{\text{echo}} \le h$.

Proof. By the same reasoning as in Lemma 6, Q will have entered round k and broadcast its own round-k proposed block B before time $T + \delta + \Delta_{\text{prop}}(h)$. Again, by the synchrony assumption and Lemma 3, B appears as a valid block in every honest party's pool before time $T + 2\delta + \Delta_{\text{prop}}(h) \le T + \Delta_{\text{ntry}}(h+1)$. This implies every honest party will not echo any blocks for parties of rank greater than h. □

Note that the condition (v) of Lemma 7 will always be satisfied by the delay functions defined in (2) when $\delta \le \Delta_{\text{bnd}}$.

We next prove a bound on the latency of a round, in terms of the rank of the leading honest party for that round. This result is proven assuming the network is δ-synchronous for a certain interval of time. However, unlike Lemmas 6 and 7, no relationship between δ and the delay functions Δ_{prop} and Δ_{ntry} is assumed.

Lemma 8 (Latency). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) at time T, all honest parties have been initiated, and k is the highest numbered round any honest party has entered;

(iii) the leading honest party Q of round k is of rank h;

(iv) the communication network is δ-synchronous at all times throughout the interval

$$[T, T + \Delta_0(h, \delta) + 2h\delta],$$

where

$$\Delta_0(h, \delta) := \max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h)). \tag{3}$$

Then the all honest parties finish round k by time T plus

$$\Delta_0(h, \delta) + (2h+1)\delta. \tag{4}$$

Proof. By the same reasoning as in Lemma 6, all honest parties will have entered round k and reached the main loop of round k by time $T + \delta$.[3] Moreover, Q will have broadcast its own round-k proposed block B by time $T + \delta + \Delta_{\text{prop}}(h)$. Again, by the synchrony assumption and Lemma 3, B appears as a valid block in every honest party's pool by time $T + 2\delta + \Delta_{\text{prop}}(h)$. Therefore, every honest party will broadcast notarization shares for B by time $T + \Delta_0(h, \delta)$, which will cause all parties to finish the round by time $T + \Delta_0(h, \delta) + \delta$, *unless some honest party has received and echoed a lower ranked block by time $T + \Delta_0(h, \delta)$.*

So we assume from now on that by time $T + \Delta_0(h, \delta)$ some honest party has received and echoed a block of rank $< h$.

[3]Unlike in Lemma 6, in this lemma, we also allow $k = 1$, but the same result holds as well in this case.

We will consider the points in time $T + \Delta_0(h, \delta) + i\delta$ for $i = 0, \ldots, 2h$. At each point in time, we will assign a status to each rank $r < h$, which is either *unused*, *used*, or *spoiled*. Each such rank will start out as *unused*. At each such point in time $T + \Delta_0(h, \delta) + i\delta$, we will change the status of at most one rank r so that the following conditions hold:

- the rank r may change from *unused* to *used*, and if this happens, at least one block of rank r has been received and echoed by some honest party by time $T+\Delta_0(h, \delta)+i\delta$, and all honest parties will have a block of rank r in its pool by time $T + \Delta_0(h, \delta) + (i+1)\delta$.

- the rank r may change from *used* to *spoiled*, and if this happens, at least two distinct blocks of rank r have been received and echoed by the honest parties by time $T + \Delta_0(h, \delta) + i\delta$, and each honest party will have two distinct blocks of rank r in its pool by time $T + \Delta_0(h, \delta) + (i + 1)\delta$.

Moreover, as we will see, if no rank changes status, then all honest parties will finish the round by time $T + \Delta_0(h, \delta) + (i + 1)\delta$. Since there can be no more than $2h$ status changes, all parties will finish the round by time $T + \Delta_0(h, \delta) + (2h + 1)\delta$.

We are assuming that at the point in time $T + \Delta_0(h, \delta)$, some honest party has received and echoed a block of rank $< h$. Choose the smallest such rank and change its status from *unused* to *used*.

We consider, in turn, the points in time $T + \Delta_0(h, \delta) + i\delta$, for $i = 1, \ldots, 2h$. Let r be the smallest *used* rank at time $T + \Delta_0(h, \delta) + (i - 1)\delta$, setting $r := h$ if there is no such rank.

Now, suppose that by time $T + \Delta_0(h, \delta) + i\delta$, some honest party has received and echoed a block of *unused* rank that is less than r; in this case, we choose the smallest such rank and change its status to *used*.

Otherwise, by time $T + \Delta_0(h, \delta) + i\delta$, all honest parties have disqualified any currently *spoiled* ranks that are less than r, and have received and echoed at least one block of rank r.

- If each party broadcast a notarization share on the same block, all honest parties will finish the round by time $T + \Delta_0(h, \delta) + (i + 1)\delta$.

- Otherwise, at least two different rank r blocks have been notarized, and we change the status of r to *spoiled*.

□

Note that the latency bound (4) in Lemma 8 measures the time between the first honest party starts round k and the last honest party finishes round k. It does not take into account the extra communication delay to actually finalize a block, if such a finalization occurs at all. This would add a term of δ to the bound (4). The bound (4) is more properly seen as controlling the *throughput* of the protocol — the rate at which blocks are notarized, which is proportional to the amortized rate at which payloads are output by the protocol (taking into account that payloads may contain many commands).

We note that Lemma 8 is essentially tight, as there is an attack that will result in a delay essentially equal to the bound in the lemma.

4.3.1 Probabilistic analysis

Let h be the rank of the leading honest party in round k, which we can view as a random variable. The results on message complexity and latency in Lemma 7 and 8 depend on the value of h. Suppose the adversary corrupts parties *statically* (i.e., at the very beginning of the protocol execution), or at least, before the random beacon at round k is revealed to the adversary. In this case, the adversary's choice of which parties to corrupt and the ranking function at round k are independent. Also assuming that $t < n/3$, it follows that for $i = 1, \ldots, t$,

$$\Pr[h \geq i] = \frac{t}{n} \cdot \frac{t-1}{n-1} \cdot \cdots \cdot \frac{t-(i-1)}{n-(i-1)} \;<\; \frac{1}{3^i}. \tag{5}$$

Of course, for $i > t$, we have $\Pr[h \geq i] = 0$. In particular, by the tail sum formula, we can bound the expected value of h as

$$\mathrm{E}[h] = \sum_{i \geq 1} \Pr[h \geq i] \leq \sum_{i \geq 1} \frac{1}{3^i} = \frac{1}{2}.$$

It follows that provided the network remains δ-synchronous throughout round k, we have $\mathrm{E}[N^{(k)}_{\mathrm{echo}}] \leq \frac{1}{2}$, and hence the expected message complexity for that round is $O(n^2)$. However, the tail bound (5) is the more important fact.

Let us now consider expected latency under the assumptions of Lemma 8. Let is assume that the delay functions Δ_{ntry} and Δ_{prop} are defined as (2). In that case, we have

$$\Delta_0(h, \delta) = 2\Delta_{\mathrm{bnd}} h + \delta + \max(\epsilon, \delta),$$

and so the latency (4) is bounded by

$$2(\Delta_{\mathrm{bnd}} + \delta)h + 2\delta + \max(\epsilon, \delta),$$

and since $\mathrm{E}[h] = 1/2$, the expected latency is at most

$$\Delta_{\mathrm{bnd}} + 3\delta + \max(\epsilon, \delta).$$

4.4 Liveness under interval synchrony

As stated, Lemma 6 guarantees liveness for a given round if the network is δ-synchronous for a short interval of time starting at the point in time at which the first honest party enters that round. It would be better to be able to say that whenever the network remains δ-synchronous for a sufficiently long interval of time, then the liveness will hold for whatever round the protocol has reached. We can obtain such a result by combining Lemma 6 with Lemma 8 on latency:

Lemma 9 (Liveness assuming interval synchrony). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) at time T, all honest parties have been initiated, and k is the highest numbered round any honest party has entered;

(iii) the leading honest party of round k is of rank h;

(iv) the communication network is δ-synchronous at all times throughout the interval

$$[T, T + \Delta_0(h, \delta) + (2h + 2)\delta + \Delta_{\text{prop}}(0)],$$

where $\Delta_0(h, \delta)$ is defined as in (3);

(v) the leader of round $k + 1$ is honest;

(vi) $2\delta + \Delta_{\text{prop}}(0) \le \Delta_{\text{ntry}}(1)$.

Then when all round-$(k+1)$ messages from honest parties have been delivered to all honest parties, each honest party will have the block proposed by the leader of round $k + 1$ in its pool as a finalized block.

If we extend the length of the interval of synchrony even further, we can guarantee that with overwhelming probability, some round within that interval will finalize a block proposed by an honest leader.

4.5 Proposal complexity

In a given round k, we define $N^{(k)}_{\text{prop}}$ to be the highest rank of any honest party that proposes a block of its own in round k. The size of $N^{(k)}_{\text{prop}}$ determines the number of distinct blocks that are circulating through the network in round k (at least, those proposed by honest parties). As discussed briefly in Section 1, bounding $N^{(k)}_{\text{prop}}$ will help control the overall communication complexity, especially when the underlying broadcasts are implemented using a peer-to-peer gossip sub-layer.

In this section, we investigate under what assumptions $N^{(k)}_{\text{prop}}$ remains well bounded.

Lemma 10 (Proposal complexity assuming partial synchrony). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) $k > 1$, the first honest party to enter round k does so at time T, and all honest parties have been initiated by time T;

(iii) the leading honest party Q of round k is of rank h;

(iv) the communication network is δ-synchronous at all times throughout the interval

$$[T, T + \Delta_0(h, \delta) + 2h\delta],$$

where $\Delta_0(h, \delta)$ is defined as in (3);

(v) $\ell \ge 1$ is such that

$$\Delta_0(h, \delta) + (2h + 1)\delta \le \Delta_{\text{prop}}(h + \ell). \quad (6)$$

Then $N^{(k)}_{\text{prop}} < h + \ell$.

This lemma follows easily from Lemma 8 on latency. Suppose the delay functions are defined as in (2), and that $\delta \leq \Delta_{\mathrm{bnd}}$ and $\epsilon \leq \Delta_{\mathrm{bnd}}$, so that

$$\Delta_0(h, \delta) \leq 2(h+1)\Delta_{\mathrm{bnd}}.$$

It follows that if the network remains δ-synchronous throughout round k, then we have

$$N^{(k)}_{\mathrm{prop}} \leq 2h + 1.$$

4.6 Analysis for crash failures

In the common case where there are only crash failures, we revisit some of the above analysis.

In this setting, if the leading honest party has rank h, then the parties of rank $0, \ldots, h-1$ are crashed.

Lemma 11 (Liveness and message complexity assuming partial synchrony and crash failures). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) $k > 1$, the first honest party to enter round k does so at time T, and all honest parties have been initiated by time T;

(iii) the leading honest party Q of round k is of rank h;

(iv) the communication network is δ-synchronous at times T and $T + \delta + \Delta_{\mathrm{prop}}(h)$;

(v) $2\delta + \Delta_{\mathrm{prop}}(h) \leq \Delta_{\mathrm{ntry}}(h+1)$.

Then

(a) when all round-k messages from honest parties have been delivered to all honest parties, each honest party will have Q's round-k proposed block in its pool as a finalized block, and

(b) $N^{(k)}_{\mathrm{echo}} \leq h$, and in fact, the only block echoed by any honest party is the block proposed by Q.

Note that the condition (v) of Lemma 11 will be satisfied by the delay functions defined in (2) when $\delta \leq \Delta_{\mathrm{bnd}}$.

Lemma 12 (Latency assuming crash failures). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) at time T, all honest parties have been initiated, and k is the highest numbered round any honest party has entered;

(iii) the leading honest party Q of round k is of rank h;

(iv) the communication network is δ-synchronous at all times throughout the interval

$$[T, T + \Delta_0(h, \delta)],$$

where $\Delta_0(h, \delta)$ is defined as in (3);

Then the all honest parties finish round k by time T plus

$$\Delta_0(h, \delta) + \delta. \tag{7}$$

Lemma 13 (Proposal complexity assuming partial synchrony and crash failures). *Assume that:*

(i) there are at most $t < n/3$ corrupt parties, signatures are secure, and hash functions are collision resistant;

(ii) $k > 1$, the first honest party to enter round k does so at time T, and all honest parties have been initiated by time T;

(iii) the leading honest party Q of round k is of rank h;

(iv) the communication network is δ-synchronous at all times throughout the interval

$$[T, T + \Delta_0(h, \delta)],$$

where $\Delta_0(h, \delta)$ is defined as in (3);

(v) $\ell \geq 1$ is such that

$$\Delta_0(h, \delta) + \delta \leq \Delta_{\text{prop}}(h + \ell). \tag{8}$$

Then $N_{\text{prop}}^{(k)} < h + \ell$.

Let us examine the consequences of Lemma 13 assuming again that the delay functions are defined as in (2), and that $\delta \leq \Delta_{\text{bnd}}$ and $\epsilon \leq \Delta_{\text{bnd}}$, so that

$$\Delta_0(h, \delta) \leq 2(h + 1)\Delta_{\text{bnd}}.$$

It follows that if the network remains δ-synchronous throughout round k, then we have

$$N_{\text{prop}}^{(k)} \leq h + 1.$$

In particular, there will be at most two blocks circulating through the network in round k, namely a block of rank h, and (possibly) a block of rank $h + 1$.

4.7 Analysis for consistent failures

It is also fruitful to consider a setting where parties may be corrupt, but behave consistently, meaning that they do not send inconsistent proposals that would disqualify them. This is especially relevant in a variation of the protocol in which parties are *permanently* disqualified, as discussed in more detail in Section 5.1. With this variation, there are at most t rounds where any party can be disqualified.

One result that can be improved is Lemma 8 on latency. In particular, the latency bound (4) can be reduced to of

$$\Delta_0(h, \delta) + (h+1)\delta, \tag{9}$$

where $\Delta_0(h, \delta)$ is defined as in (3).

Another result that can be improved is proposal complexity. Specifically, the condition (6) in Lemma 10) can be replaced by the somewhat weaker condition

$$\Delta_0(h, \delta) + (h+1)\delta \leq \Delta_{\text{prop}}(h + \ell). \tag{10}$$

4.8 Local delay functions

So far, for simplicity, we have assumed that all honest parties use the same delay functions Δ_{prop} and Δ_{ntry} in every round. This is not entirely realistic, for at least two reasons.

First, the honest parties' clocks may run at slightly different rates (clock drift). This can be modeled by having different delay functions for each party.

Second, if the network delay may vary over time, honest parties may wish to dynamically adjust their delay functions to adjust for this, based on the perceived performance of the system.

For example, if a party sees too may rounds go by without a finalization, or sees unexpectedly high message complexity for several rounds, it may increase its Δ_{ntry} delay function. After such an increase in Δ_{ntry}, a party may attempt to reduce it at a later round. For the delay functions defined in (2), these adjustments to Δ_{ntry} could be made by simply adjusting the value of Δ_{bnd} used in defining the function Δ_{ntry} (without necessarily changing the value of Δ_{bnd} used in defining the function Δ_{prop}). In another scenario, a party may increase the additive term ϵ in (2) to slow the protocol down if it is running too fast relative to other elements in the system. In any case, each party is making these decisions locally, and so they will invariably end up with different delay functions.

Because of this, we introduce **local delay functions**. Specifically, let us assume that in round k party Q uses the delay functions $\Delta_{\text{prop}}^{(k,Q)}$ and $\Delta_{\text{ntry}}^{(k,Q)}$. We assume that for each k and Q, the functions $\Delta_{\text{prop}}^{(k,Q)}$ and $\Delta_{\text{ntry}}^{(k,Q)}$ are non-decreasing.

For each k, we define

$$\underline{\Delta}_{\text{prop}}^{(k)} := \min\{\Delta_{\text{prop}}^{(k,Q)} : Q \text{ honest}\} \quad \text{and} \quad \overline{\Delta}_{\text{prop}}^{(k)} := \max\{\Delta_{\text{prop}}^{(k,Q)} : Q \text{ honest}\},$$

and similarly,

$$\underline{\Delta}_{\text{ntry}}^{(k)} := \min\{\Delta_{\text{ntry}}^{(k,Q)} : Q \text{ honest}\} \quad \text{and} \quad \overline{\Delta}_{\text{ntry}}^{(k)} := \max\{\Delta_{\text{ntry}}^{(k,Q)} : Q \text{ honest}\}.$$

One can see that for each k, each of the functions

$$\underline{\Delta}^{(k)}_{\text{prop}},\ \overline{\Delta}^{(k)}_{\text{prop}},\ \underline{\Delta}^{(k)}_{\text{ntry}},\ \text{and}\ \overline{\Delta}^{(k)}_{\text{ntry}}$$

is non-decreasing.

We now revisit each of our main results that depend on these delay functions. As will become evident, to maintain our liveness and message complexity results, it is important that while parties may locally increase their Δ_{ntry} functions, they should *not* locally adjust their Δ_{prop} functions. We may nevertheless still model Δ_{prop} locally, if only to properly take into account clock drift.

Liveness. Lemma 6 is still true, but with condition (v) replaced by

$$2\delta + \overline{\Delta}^{(k)}_{\text{prop}}(0) \le \underline{\Delta}^{(k)}_{\text{ntry}}(1). \tag{11}$$

Similarly, in the setting of crash failures, the liveness result (part (a)) of Lemma 11 is still true with condition (v) of that lemma replaced by

$$2\delta + \overline{\Delta}^{(k)}_{\text{prop}}(h) \le \underline{\Delta}^{(k)}_{\text{ntry}}(h+1). \tag{12}$$

In terms of preserving liveness, there is no benefit in making any local adjustments to the function Δ_{prop}, and indeed, locally increasing the value of Δ_{prop} would only make it less likely for (11) and (12) to be satisfied. Thus, let us assume from now on that there are no local adjustments made to the function Δ_{prop} — indeed, there is no reason ever to use a value of $\Delta_{\text{prop}}(0)$ other than 0, so we may assume $\overline{\Delta}^{(k)}_{\text{prop}}(0) = 0$.

Now, if several rounds go by without finalization, and each party Q heuristically increases $\Delta^{(k,Q)}_{\text{ntry}}$ enough to compensate for a network delay that is larger than expected, then finalization will resume in round k. Moreover, as far as liveness is concerned, there is no penalty if $\overline{\Delta}^{(k)}_{\text{ntry}}$ is excessively large, in the sense that (11) and (12) will still hold. That is, there is no penalty if the local adjustments made to Δ_{ntry} are out of step with one another, as long as they are all large enough.

Message complexity. Lemma 7 is still true with condition (v) replaced by

$$2\delta + \overline{\Delta}^{(k)}_{\text{prop}}(h) \le \underline{\Delta}^{(k)}_{\text{ntry}}(h+1). \tag{13}$$

So the same comments that apply to liveness apply to message complexity. In terms of preserving message complexity, there is no benefit in making any local adjustments to the function Δ_{prop}, and we continue to assume that no such adjustments are made. Moreover, if each party Q locally increases the function Δ_{ntry} enough to compensate for a network delay that is larger than expected, then message complexity should become bounded again in round k. In addition, as far as message complexity is concerned, there is no penalty if the local adjustments made to the function Δ_{ntry} are out of step with one another, as long as they are all large enough.

Latency. Lemma 8 is still true with the value of $\Delta_0(h, \delta)$ replaced throughout by

$$\overline{\Delta}_0^{(k)}(h, \delta) \coloneqq \max(2\delta + \overline{\Delta}_{\text{prop}}^{(k)}(h), \delta + \overline{\Delta}_{\text{ntry}}^{(k)}(h)). \tag{14}$$

It is precisely here that we incur a penalty if $\overline{\Delta}_{\text{ntry}}^{(k)}(h)$ is too large. However, we stress that regardless of $\overline{\Delta}_{\text{ntry}}^{(k)}$, the protocol is still *responsive*, i.e., runs at network speed, assuming $\overline{\Delta}_{\text{prop}}^{(k)}(0)$ and $\overline{\Delta}_{\text{ntry}}^{(k)}(0)$ are (close to) zero (which is recommended) and assuming the leader in round k is honest.

We do note, however, that we also incur the same penalty in Lemma 9 on liveness under an interval synchrony assumption — the length of the interval of time under which synchrony must hold increases as $\overline{\Delta}_{\text{ntry}}^{(k)}(h)$ increases.

Proposal complexity. Lemma 10 is still true with the value of $\Delta_0(h, \delta)$ replaced throughout by $\overline{\Delta}_0^{(k)}(h, \delta)$, as defined above in (14), and the inequality (6) replaced by

$$\overline{\Delta}_0^{(k)}(h, \delta) + (2h + 1)\delta \le \underline{\Delta}_{\text{prop}}^{(k)}(h + \ell). \tag{15}$$

As already mentioned above, to maintain liveness and message complexity, we should not locally adjust the Δ_{prop} function. So assuming $\Delta_{\text{prop}}^{(k,Q)} = \Delta_{\text{prop}}$ for all k and Q, the condition (15) then becomes

$$\max(\delta + \Delta_{\text{prop}}(h), \overline{\Delta}_{\text{ntry}}^{(k)}(h)) + (2h + 2)\delta \le \Delta_{\text{prop}}(h + \ell). \tag{16}$$

However, this reveals a tension between the goals of maintaining message complexity and proposal complexity. If the network is unexpectedly slow, and $\overline{\Delta}_{\text{ntry}}^{(k)}(h)$ is unexpectedly large, the condition (16) may be impossible to satisfy for any reasonably small ℓ.

Note that this same problem exists even if we only consider *consistent failures* or *crash failures*. In the setting of crash failures, the condition (16) in Lemma 13 becomes

$$\max(\delta + \Delta_{\text{prop}}(h), \overline{\Delta}_{\text{ntry}}^{(k)}(h)) + 2\delta \le \Delta_{\text{prop}}(h + \ell), \tag{17}$$

which also may be impossible to satisfy for any reasonably small ℓ.

One possible mitigation for this problem is to define delay functions that grow super-linearly, rather than linearly as in (2) — this would make it more likely for (17) to be satisfied for smaller values of ℓ.

5 Protocol variations

In this section, we consider a few protocol variations, and their consequences.

5.1 Permanent disqualification of inconsistent parties

As we have presented it, one party may disqualify another party if the first party detects that the second has proposed two different blocks in one round. However, that disqualification does not carry over to subsequent rounds.

```
(c) a valid round-$k$ block $B$ of rank $r$, such that
        $r \notin \text{ranks}(\mathcal{D}_\text{p})$, $\text{clock}() \geq t_0 + \Delta_\text{ntry}(r)$, and
        there is no valid round-$k$ block $B^*$ of rank $r^* \in [r] \setminus \text{ranks}(\mathcal{D}_\text{p})$, and
        ($B \notin \mathcal{N}$ or there is evidence of inconsistent behavior for rank $r$):
    if there is evidence of inconsistent behavior for rank $r$ then
        // Disqualify the party of rank $r$
        **broadcast** an inconsistency proof against the party of rank $r$
        add the rank $r$ party's index to $\mathcal{D}_\text{p}$
    else
        **broadcast** $B$, $B$'s authenticator, and the notarization for $B$'s parent
        $\mathcal{N} \leftarrow \mathcal{N} \cup \{B\}$, **broadcast** a notarization share for $B$
```

Figure 3: Logic for permanent disqualification

One can modify the protocol so that if one party disqualifies a party in some round, instead of broadcasting the second block that caused the disqualification (as our current protocol does), it broadcasts a special message called an "inconsistency proof", which proves that a party authenticated two different blocks in the same round.

Recall that in Section 3.4, we defined an authenticator for a round-k block

$$B = (\mathsf{block}, k, \alpha, \mathit{phash}, \mathit{payload})$$

as a tuple $(\mathsf{authenticator}, k, \alpha, H(B), \sigma)$, where σ is a valid S_auth-signature on $(\mathsf{authenticator}, k, \alpha, H(B))$ by party P_α.

A **pair of inconsistent authenticators from P_α for round k** is a pair of tuples

$$(\mathsf{authenticator}, k, \alpha, \mathit{hash}_1, \sigma_1), (\mathsf{authenticator}, k, \alpha, \mathit{hash}_2, \sigma_2),$$

such that σ_i is a valid S_auth-signature on $(\mathsf{authenticator}, k, \alpha, \mathit{hash}_i)$ by party P_α for $i = 1, 2$, and $\mathit{hash}_1 \neq \mathit{hash}_2$.

An **inconsistency proof against P_α for round k** is a tuple of the form

$$(\mathsf{inconsistent}, k, \alpha, \mathit{hash}_1, \sigma_1, \mathit{hash}_2, \sigma_2),$$

where σ_i is a valid S_auth-signature on $(\mathsf{authenticator}, k, \alpha, \mathit{hash}_i)$ by party P_α for $i = 1, 2$, and $\mathit{hash}_1 \neq \mathit{hash}_2$. Assuming S_auth-signatures are secure, no inconsistency proof against an honest party can be constructed. Note that with this new syntax for authenticators, such an inconsistency proof can be validated without knowing the blocks themselves.

The Tree Building Subprotocol is modified as follows. First, instead of maintaining a set $\mathcal{D}$ of disqualified ranks that is initialized to $\emptyset$ every round, each party P_α maintains a set $\mathcal{D}_\text{p}$ of parties that have been *permanently disqualified*. The set $\mathcal{D}_\text{p}$ is initialized to $\emptyset$ at the very beginning of the protocol.

Second, the logic of "wait for" condition (c) in the main loop of Figure 1 is modified as shown in Figure 3.

Here, ranks($\mathcal{D}_{\mathrm{p}}$) denotes the set of ranks for the current round of the parties in the set $\mathcal{D}_{\mathrm{p}}$. Also, if P_β is the party of rank r in the current round, then **evidence of inconsistent behavior for rank r** means either

(i) a pair of inconsistent authenticators from P_β for the *current* round, or

(ii) an inconsistency proof against P_β for the current round, or *any previous round*;

In case (ii), P_α broadcasts the inconsistency proof against P_β that it has, while in case (i), P_α constructs and broadcasts a new inconsistency proof against P_β, derived from the inconsistent authenticators.

For this variation, all of the lemmas we proved above hold without change, and there are no significant downsides. In this variation, there are at most t rounds where any party may be disqualified, and so in terms of performance of the system of the long term, it is really only the performance of the protocol where all parties behave consistently that matters. We have discussed in Section 4.7 how latency and proposal complexity improves in this setting.

5.1.1 An alternative implementation of temporary disqualification

Even if one does not wish to permanently disqualify parties, the use of inconsistency proofs and this alternative syntax for authenticators can be somewhat more practical, as it does not require broadcasting an entire second block in order to convince others parties that a party is acting inconsistently. Indeed, our original protocol with temporary disqualification could be more efficiently implemented by simply changing the definition of *evidence of inconsistent behavior for rank r* above to mean either

(i) a pair of inconsistent authenticators from P_β for the *current* round, or

(ii) an inconsistency proof against P_β for the *current* round,

5.2 Protocol ICC1: tightening the proposal condition

In presenting this variation, which we call Protocol ICC1, we will assume we have already incorporated the "permanent disqualification" rule introduced in Section 5.1. However, it can also be implemented using "temporary disqualification", with some very small changes (discussed below).

Recall that party P_α will propose its own block when $\Delta_{\mathrm{prop}}(r_{\mathrm{me}})$ time units have elapsed since the beginning of the round (actually, since the time at which it obtained the random beacon for the current round). In this variation, we will tighten the condition under which a party P_α will propose its own block, so that it will "hold back" from doing so if there is an apparently "better" block in its pool. Here, by a "better" block, we mean a block of lower rank than that of P_α that has not been disqualified.

Note, however, if a party "holds back" from making its own proposal because it sees a "better" block in its own pool, there is no guarantee that any other honest party sees this "better" block any time soon. So instead of simply "holding back", the party will echo this "better" block. Somewhat more precisely, until the round is over, it will echo any "better" block of least rank r but only if $\Delta_{\mathrm{prop}}(r)$ time units have elapsed since the beginning of the round.

The motivation for this variation is to minimize bandwidth utilization in the setting where the underlying communication layer is a peer-to-peer gossip layer (as in the Internet Computer's implementation). In this setting, what really want to minimize is the total number of *distinct* blocks (which may be very large) that are circulating through the network, rather than to minimize message complexity (or communication complexity) as traditionally defined. Moreover, this variation is optimized for the crash failure and consistent failure settings (which are the most relevant if the protocol permanently disqualifies inconsistent parties, either using the mechanism here, or using some other mechanism higher up in the protocol stack). This protocol variation most closely resembles the protocol as currently designed for the Internet Computer. We will briefly discuss more connections between this protocol variation and the peer-to-peer gossip layer below (see Section 5.2.1).

The details of this variation are shown in Figure 4. Note that to simplify the logic of the protocol, we also assume in this section that $\Delta_{\text{prop}}(r) \leq \Delta_{\text{ntry}}(r)$ for all $r \in [n]$. In the setting with local delay functions (as in Section 4.8), this inequality is only required to hold locally (i.e., $\Delta_{\text{prop}}^{(k,Q)}(r) \leq \Delta_{\text{ntry}}^{(k,Q)}(r)$ for each k and Q). Here, $\mathcal{D}_{\text{p}}$ is the set of permanently disqualified parties, and ranks($\mathcal{D}_{\text{p}}$) is the set of ranks of these parties, as determined by the random beacon for the current round. The details of the notions of "evidence of inconsistent behavior" and "inconsistency proof" can be found in Section 5.1. Note that if we wish to implement temporary disqualifications instead of permanent disqualifications, we can do so by simply (a) resetting $\mathcal{D}_{\text{p}}$ to $\emptyset$ at the start of each round, and (b) defining the notions of "evidence of inconsistent behavior" and "inconsistency proof" as in Section 5.1.1.

5.2.1 More on the peer-to-peer gossip layer

Without going into too many details, we can say a few more words about the relationship between this protocol variation and the underlying peer-to-peer gossip layer, as implemented by the Internet Computer.

Gossip communication is sometimes described as satisfying the following properties:

- If any honest party *sends* a message, then eventually (or in a timely manner, under a partial synchrony assumption), every honest party receives the message.
- If any honest party *receives* an "interesting" message, then eventually (or in a timely manner, under a partial synchrony assumption), every honest party receives the message.

Clearly, for this type of communication to remain bounded at all, the criteria used for what constitutes an "interesting" message has to be fairly restrictive. Indeed, if all messages are "interesting", then the communication complexity may become completely unbounded.

In the protocol variation in this section, in terms of the peer-to-peer gossip layer, a party views a block as "interesting", and is willing to tell other parties about it, if it is a notarized block, or an unnotarized block that it chooses to broadcast. Thus, the number of "interesting" blocks per round circulating through the gossip layer will be finite, and as we will argue below, under a partial synchrony assumption, will be very small (at least for crash and consistent failures). Besides blocks, there are other types of messages, such as notarization shares, notarizations, etc. Generally speaking, these types of messages are

```
broadcast a share of the round-1 random beacon
$\mathcal{D}_\text{p} \leftarrow \emptyset$    // the set of parties disqualified by $P_\alpha$
For each round $k = 1, 2, 3 \ldots$ :
    wait for $t+1$ shares of the round-$k$ random beacon
    compute the round-$k$ random beacon (which defines the permutation $\pi$ for round $k$)
    broadcast a share of the random beacon for round $k+1$

    let $r_\text{me}$ be the rank of $P_\alpha$ according to the permutation $\pi$
    $\mathcal{B} \leftarrow \emptyset$    // the set of blocks that have been broadcast by $P_\alpha$
    $\mathcal{N} \leftarrow \emptyset$    // the set of blocks for which notarization shares have been broadcast by $P_\alpha$

    $\mathit{done} \leftarrow \mathsf{false}$,  $\mathit{proposed} \leftarrow \mathsf{false}$,  $t_0 \leftarrow \text{clock}()$

    repeat
        wait for either:
            (a)  a notarized round-$k$ block $B$,
                   or a full set of notarization shares for some valid
                   but non-notarized round-$k$ block $B$:
                 // Finish the round
                 combine the notarization shares into a notarization for $B$, if necessary
                 broadcast the notarization for $B$
                 $\mathit{done} \leftarrow \mathsf{true}$
                 if $\mathcal{N} \subseteq \{B\}$ then broadcast a finalization share for $B$

            (b)  a valid round-$k$ block $B$ of rank $r$, such that
                   $r \in [r_\text{me}] \setminus \text{ranks}(\mathcal{D}_\text{p})$, $\text{clock}() \geq t_0 + \Delta_\text{prop}(r)$,
                   there is no valid round-$k$ block $B^*$ of rank $r^* \in [r] \setminus \text{ranks}(\mathcal{D}_\text{p})$, and
                   ($B \notin \mathcal{B}$  or  there is evidence of inconsistent behavior for rank $r$):

                 if there is evidence of inconsistent behavior for rank $r$ then
                     // Disqualify the party of rank $r$
                     broadcast an inconsistency proof against the party of rank $r$
                     add the rank $r$ party's index to $\mathcal{D}_\text{p}$
                 else
                     // Echo block $B$
                     broadcast $B$, $B$'s authenticator, and the notarization for $B$'s parent
                     $\mathcal{B} \leftarrow \mathcal{B} \cup \{B\}$

            (c)  not $\mathit{proposed}$, $\text{clock}() \geq t_0 + \Delta_\text{prop}(r_\text{me})$, and
                   there is no valid round-$k$ block $B^*$ of rank $r^* \in [r_\text{me}] \setminus \text{ranks}(\mathcal{D}_\text{p})$:
                 // Propose a block
                 choose a notarized round-$(k-1)$ block $B_\text{p}$
                 $\mathit{payload} \leftarrow \mathit{getPayload}(B_\text{p})$
                 create a new round-$k$ block $B = (\mathsf{block}, k, \alpha, H(B_\text{p}), \mathit{payload})$
                 broadcast $B$, $B$'s authenticator, and the notarization for $B$'s parent
                 $\mathcal{B} \leftarrow \mathcal{B} \cup \{B\}$
                 $\mathit{proposed} \leftarrow \mathsf{true}$

            (d)  a block $B \in \mathcal{B} \setminus \mathcal{N}$ of rank $r \notin \text{ranks}(\mathcal{D}_\text{p})$ such that
                   $\text{clock}() \geq t_0 + \Delta_\text{ntry}(r)$,
                   there is no valid round-$k$ block $B^*$ of rank $r^* \in [r] \setminus \text{ranks}(\mathcal{D}_\text{p})$, and
                   there is no evidence of inconsistent behavior for rank $r$:
                 // Generate notarization share for $B$
                 $\mathcal{N} \leftarrow \mathcal{N} \cup \{B\}$, broadcast a notarization share for $B$
    until $\mathit{done}$
```

Figure 4: ICC1: Tree Building Subprotocol for party P_α

associated with a particular block, and will be deemed “interesting” if the block itself is interesting.

Besides parties that may be proposing and notarizing blocks, the network may contain additional “relay” nodes to help spread “interesting” messages. Such relay nodes essentially run the protocol as if they had rank ∞ — in particular, they never propose their own blocks, nor do they ever generate any notarization, finalization, or random beacon shares. However, such nodes can choose to echo blocks, using the same rules as for ordinary participants. Moreover, such “relay” nodes may also be running their own copies of the replicated state machine.

For large messages, if a node has a message it thinks is “interesting”, it will send an “advert” to its immediate peers in the network. Such an “advert” is a very compact description of the actual message. For example, for a block, such an “advert” may contain all of the information about the block, except for the payload itself, which is usually quite large. If a peer sees the “advert” and also thinks that the corresponding message would be “interesting” (e.g., a block that it would itself echo), then the peer will request the message itself. The requesting peer may also prioritize such requests, based on which messages seem more “interesting” (in a given round, a lower-ranked block is typically more “interesting”). Such heuristics generally yield good bandwidth utilization, while still maintaining liveness under reasonable assumptions (and, of course, safety is never jeopardized).

5.2.2 Analysis

Recall that we defined

- $N^{(k)}_{\text{prop}}$ is the highest ranked honest party that proposes its own block in round k, and
- $N^{(k)}_{\text{echo}}$ is the highest ranked block echoed by any party.

Let us define

$$N^{(k)}_{\max} := \max(N^{(k)}_{\text{prop}}, N^{(k)}_{\text{echo}}).$$

This is simply the highest rank block that is broadcast by any honest party in round k, under any circumstances.

Let us explore the consequences of the stricter block proposing conditions of ICC1. Lemmas 1–6 (including safety and liveness) still hold. Also, in terms of liveness (but not message complexity) in the crash failure setting, the stronger result of part (a) Lemma 11 holds. Note that these results on liveness also hold in the local delay function setting, as in Section 4.8. Just as we discussed in that section, if each party locally increases the function Δ_{ntry} sufficiently (but does not locally adjust Δ_{prop}), liveness will be achieved. Moreover, it does not matter if the local delay functions Δ_{ntry} are out of sync — all that matters is that each of them is large enough.

As discussed above, in terms of bandwidth utilization, we are mainly interested in bounding the number of distinct blocks that are circulating through the network in the crash failure and consistent failure settings. So in this setting, this is equivalent to bounding the value of $N^{(k)}_{\max}$.

Bounding $N_{\max}^{(k)}$ in the crash and consistent failure settings. In either the crash failure or consistent failure setting, if h is the rank of the leading honest party in round k, and the network is δ-synchronous in that round, then we will have $N_{\max}^{(k)} < h + \ell$ provided

$$2\delta + \Delta_{\text{prop}}(h) \leq \Delta_{\text{prop}}(h + \ell). \tag{18}$$

Proof sketch. If the first honest party enters the round at time T, the party Q of rank h will enter before time $T + \delta$, and will either broadcast its own proposal or a lower ranked proposal before time $T + \delta + \Delta_{\text{prop}}(h)$. This will arrive at all honest parties before time $T + 2\delta + \Delta_{\text{prop}}(h)$. Since there are no disqualifications, that proposal will prevent any honest party from broadcasting any proposal of rank higher than h. □

If we define the delay functions linearly as in (2), and $\delta \leq \Delta_{\text{bnd}}$, this condition will hold with $\ell \coloneqq 1$. In the crash failure setting, this means that there will be only one block in circulation. In the consistent failure setting, there could be up to $h+1$ blocks in circulation.

If the network delay bound Δ_{bnd} in (2) is too small, then we will need to choose a larger value of ℓ in order to satisfy (18), and hence $N_{\max}^{(k)}$ may be larger. This can be mitigated by delay functions that grow super-linearly, rather than linearly. Note, however, that as Δ_{prop} appears on both sides of the inequality (18), locally adjusting the delay functions as in Section 4.8 will not be very helpful in mitigating this.

Let us compare this bound on $N_{\max}^{(k)}$ to our original protocol. For that protocol, in order to ensure that $N_{\max}^{(k)} < h + \ell$, we need

$$\begin{aligned} 2\delta + \Delta_{\text{prop}}(h) &\leq \Delta_{\text{ntry}}(h + \ell), \text{ and} \\ \max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h)) + (h+1)\delta &\leq \Delta_{\text{prop}}(h + \ell) \end{aligned} \tag{19}$$

in the consistent failure setting, and

$$\begin{aligned} 2\delta + \Delta_{\text{prop}}(h) &\leq \Delta_{\text{ntry}}(h + \ell), \text{ and} \\ \max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h)) + \delta &\leq \Delta_{\text{prop}}(h + \ell) \end{aligned} \tag{20}$$

in the crash failure setting. One can see that condition (18) is strictly weaker than either of conditions (19) and (20). Moreover, if we locally adjust the delay function Δ_{ntry}, conditions (19) and (20) become even harder to satisfy. Thus, in terms of bounding $N_{\max}^{(k)}$, in the consistent failure and crash failure settings, this variation is superior.

Bounding $N_{\max}^{(k)}$ in the Byzantine failure setting. Let us also consider the Byzantine setting. In this setting, while the quantity $N_{\max}^{(k)}$ regulates the message complexity of this protocol variation, it no longer regulates the number of distinct blocks circulating through the network (since now there may be several blocks of a given rank). In this setting, if h is the rank of the leading honest party in round k, and the network is δ-synchronous in that round, then we will have $N_{\max}^{(k)} < h + \ell$ provided

$$\max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h - 1)) + 2h\delta \leq \Delta_{\text{prop}}(h + \ell). \tag{21}$$

Proof sketch. If the first honest party enters the round at time T, the party Q of rank h will enter before time $T + \delta$, and will either broadcast its own proposal or a lower ranked proposal before time $T + \delta + \Delta_{\text{prop}}(h)$.

There are two cases to consider. In the first case, Q broadcasts its own proposal before time $T + \delta + \Delta_{\text{prop}}(h)$, and the argument follows as above.

In the second case, Q broadcasts a proposal of lower rank, and holds back from broadcasting its own. We want to argue that before time

$$T + \overbrace{\max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h-1))}^{\Delta_0' :=} + 2h\delta,$$

either all honest parties have finished the round or have received Q's proposal, which will prevent any honest party from broadcasting any proposal of rank higher than h.

Now, before time $T + \Delta_0'$, all honest parties have received a lower ranked proposal from Q and are ready to issue a notarization share on it. We make a status-changing argument as in the proof of Lemma 8, where each of the ranks $0, \ldots, h-1$ is assigned a status of *unused*, *used*, or *spoiled*. However, before we start the status-changing argument at time $T + \Delta_0'$, we may already initialize the rank of the first proposal broadcast by Q the status of *used*, rather than *unused*. By the same reasoning used in Lemma 8, after at most $2h - 1$ status changes, so by time $T + \Delta_0' + (2h-1)\delta$, either

(a) all honest parties have issued a notarization share on the same block, or

(b) Q will have broadcast its own proposal.

So before time $T + \Delta_0' + 2h\delta$, either all honest parties have finished the round or have received Q's proposal. □

Let us compare this bound on $N_{\max}^{(k)}$ to our original protocol. For that protocol, then in order to ensure that $N_{\max}^{(k)} < h + \ell$, we need

$$\begin{aligned} 2\delta + \Delta_{\text{prop}}(h) &\le \Delta_{\text{ntry}}(h+\ell), \text{ and} \\ \max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h)) + (2h+1)\delta &\le \Delta_{\text{prop}}(h+\ell). \end{aligned} \tag{22}$$

Condition (22) implies condition (21). Thus, in terms of bounding $N_{\max}^{(k)}$, in the Byzantine setting, this variation is also superior. However, note that both the original protocol and this variant will have trouble bounding $N_{\max}^{(k)}$ in the Byzantine failure setting if the function Δ_{ntry} is locally adjusted.

Latency. All of our results on latency hold without change, in the crash, consistent, and Byzantine failure settings.

Proof sketch. For crash failures, nothing changes.

For Byzantine failures, we argue as follows. If the first honest party enters the by time T, the party Q of rank h will enter before time $T + \delta$, and will either broadcast its own proposal or a lower ranked proposal before time $T + \delta + \Delta_{\text{prop}}(h)$.

There are two cases to consider. In the first case, Q broadcasts its own proposal by time $T + \delta + \Delta_{\text{prop}}(h)$, and the argument follows as in the analysis of the original protocol.

As we argued above in analyzing $N_{\text{max}}^{(k)}$, by time

$$T + \overbrace{\max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h-1))}^{\Delta_0' :=} + s\delta,$$

where $s \leq 2h - 1$, Q will have either broadcast its own proposal or all honest parties will have broadcast a notarization share on the same block, and we will have made $s + 1$ status changes. So by time

$$T + \max((s+1)\delta + \Delta_0', \delta + \Delta_{\text{ntry}}(h))$$

all honest parties will have either finished the round, or be ready to notarize Q's proposal. In the latter case, they will all notarize Q's proposal, unless there are more status changes, of which there can be at most $2h - (s + 1)$. So by time

$$T + \max((s+1)\delta + \Delta_0', \delta + \Delta_{\text{ntry}}(h)) + (2h - (s+1))\delta,$$

all parties will notarize Q's proposal, and all parties will finish the round by time T plus

$$\max((s+1)\delta + \Delta_0', T + \delta + \Delta_{\text{ntry}}(h)) + (2h - s)\delta,$$

which one can verify is at most the delay bound (4).

For consistent failures, we argue as follows. If the first honest party enters the by time T, the party Q of rank h will enter by time $T + \delta$, and will either broadcast its own proposal or a lower ranked proposal by time $T + \delta + \Delta_{\text{prop}}(h)$.

There are two cases to consider. In the first case, Q broadcasts its own proposal by time $T + \delta + \Delta_{\text{prop}}(h)$, and the argument follows as in the analysis of the original protocol.

Using an argument similar to that used above, and using the fact that parties are consistent, by time

$$T + \overbrace{\max(2\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h-1))}^{\Delta_0' :=} + (h-1)\delta,$$

all honest parties will have broadcast a notarization share on the same block. So all parties finish the round by time T plus

$$T + \Delta_0' + h\delta,$$

which one can verify is less than the delay bound (9). $\square$

5.3 Protocol ICC2: reducing the communication bottleneck

In Protocol ICC1 in Section 5.2, much of the protocol, especially the logic in condition (b) of the "wait for" in the main loop, can be seen as running a number of simple broadcast subprotocols in parallel, but with staggered start times — a party of rank r_{me} is only willing to participate a broadcast subprotocol a machine of rank $r < r_{\text{me}}$ if $\Delta_{\text{prop}}(r)$ time units have passed since the beginning of the round.

However, the underlying broadcast subprotocol used does not by itself guarantee consistency, which is why the atomic broadcast protocol itself needs to have additional logic to disqualify inconsistent parties. Moreover, the underlying broadcast subprotocol is not optimal in terms of communication complexity — at least under the traditional metric of communication complexity, where we simply count the number of bits sent by all honest parties. If blocks have size S, then ignoring the communication complexity contributed by signatures and signature shares (which can be orders of magnitude smaller than large blocks), the communication complexity is $O(n^2 S)$.

In this section, we sketch how we can substitute the underlying broadcast subprotocol by a *reliable broadcast protocol*, which will eliminate the need to disqualify inconsistent parties. Moreover, the particular reliable broadcast protocol we use has a communication complexity of just $O(nS)$, under the assumption that $S = \Omega(n \log n \, \lambda)$, and that signatures and hashes have length $O(\lambda)$.

One downside to this approach is that the best-case latency per round increases somewhat — but only by one network delay δ. The only other downside to this approach is that the computational complexity of the protocol is somewhat higher.

5.3.1 Reliable broadcast

In a *reliable broadcast protocol*, we have parties $P_1, \ldots, P_n$, of which $t < n/3$ may be corrupt, and we assume an asynchronous communication network. In each round k,[4] each party P_α may initiate the protocol to reliably broadcast a single message m to all parties: we say P_α *initiates reliable broadcast from P_α in round k with message m.* In each round k, each party P_β may initiate the protocol to reliably receive some message from P_α, for $\alpha \neq \beta$, in round k: we say P_β *initiates reliable broadcast from P_α in round k.* In each round k, each party P_β that has initiated reliable broadcast from P_α in round k (where we may have $\alpha = \beta$) may later output a single message m: we say P_β *reliably receives m from P_α in round k.*

The key properties of such a protocol are the following, which should hold for each round k:

Validity: if an honest party P_α initiates reliable broadcast from P_α in round k with message m, then every honest party eventually reliably receives m from P_α in round k.

Consistency: if two honest parties reliably receive messages m and m', respectively, from a party P_α in round k, then $m = m'$.

Integrity: if an honest party reliably receives a message m from an honest party P_α in round k, then P_α previously initiated reliable broadcast from P_α in round k on m.

Totality: if one honest party reliably receives a message from P_α in round k, then eventually each honest party does so.

For the validity and totality properties, “eventually” means when all honest parties have initiated the protocol (either as a broadcaster or receiver) for reliable broadcast from P_α in round k, and all messages generated by honest parties associated with this reliable broadcast instance have been delivered.

[4]Because of our application, we think of k as a round number. More generally, k could be any type of “session identifier”.

5.3.2 Reliable broadcast with reduced communication complexity using erasure codes

We sketch a reliable broadcast protocol that is based on an erasure code, which is a variation on the protocol in [CT05] (similar techniques were used in [AKK$^+$02, AKK$^+$07]). While the protocol in [CT05] can be seen as being derived from Bracha's broadcast protocol [Bra87], the protocol we present here can be seen as being derived from the simpler protocol presented in Fig. 1 of [ANRX21], and it features a lower latency than the one in [CT05]. Our protocol also has one extra property that is essential in the our atomic broadcast protocol, which we call *strong totality*, and is discussed below. In principle, one could use any reliable broadcast protocol in our atomic broadcast protocol that satisfies this extra property.

The messages that we will wish to reliably broadcast are blocks, and so we will restrict ourselves to that setting.

We shall need an $(n, n-2t)$ erasure code (see, for example, [Riz97], for a brief introduction to erasure codes, further references, as well as applications of such codes in the context of network protocols). Such a code can take a block B of size S and break it up into n fragments, each of size $\approx S/(n-2t)$, with the property that any $n-2t$ fragments can be used to reconstruct the block B. Note that under the assumption that $t < n/3$, we have $S/(n-2t) < 3S/n$, and so all n fragments together are of size at most $\approx 3S$.

We shall also need a Merkle tree [Mer80, Mer89]. A party who knows a block B can take all n fragments of B, and construct a Merkle tree with these n fragments as leaves, and publish just the root of the Merkle tree. That same party can verifiably publish one fragment of B by publishing the fragment and the nodes (and their siblings) along the corresponding path in the Merkle tree. We call these nodes the *validating path*.

Finally, we need an instance S_{rbc} of a $(t, n-t, n)$-threshold signature scheme.

When P_α reliably broadcasts its own proposed block B in round k, it first computes

- $\mathit{hash} \leftarrow H(B)$,
- fragments $F_1, \ldots, F_n$ of B, and
- a Merkle tree with $F_1, \ldots, F_n$ as leaves.

It then constructs an *authenticator* for B, which now has a different syntax from that in Section 3.4: it is a tuple $(\mathsf{authenticator}, k, \alpha, \mathit{hash}, \sigma)$, where *hash* is the root of the computed Merkle tree, and σ is a valid S_{auth}-signature on $(\mathsf{authenticator}, k, \alpha, \mathit{hash})$ by party P_α. More generally:

> *throughout our atomic broadcast protocol, instead of using the hash of a block as a "handle" for a block, we shall use the root of the corresponding Merkle tree instead; this applies not only to authenticators, but also to notarizations and finalizations, and to identify the parent of a block within a block proper.*

Party P_α will also construct a *verifiable fragment* for each party P_β, which is of the form

$$(\mathsf{rbc\text{-}fragment}, k, \alpha, \mathit{hash}, \beta, \mathit{path}, F_\beta),$$

where *path* is the validation path for the leaf F_β in the Merkle tree. Finally, party P_α

- sends to each P_β its corresponding fragment,
- broadcasts B's authenticator and the notarization of B's parent to all parties,
- *reliably receives* B.

Now consider the logic for a party P_β that has initiated reliable broadcast from P_α in round k, which includes the case where $\alpha = \beta$.

1. If and when P_β receives both a valid authenticator

   $$(\textsf{authenticator}, k, \alpha, \mathit{hash}, \sigma)$$

   and a valid corresponding verifiable fragment

   $$(\textsf{rbc-fragment}, k, \alpha, \mathit{hash}, \beta, \mathit{path}, F_\beta),$$

   it will broadcast that verifiable fragment to all parties, along with an *rbc-validation share*, which is of the form

   $$(\textsf{rbc-validation-share}, k, \alpha, \mathit{hash}, \mathit{rvs}, \beta),$$

   where *rvs* is a valid S_{rbc}-signature share by party P_β on

   $$(\textsf{rbc-validation}, k, \alpha, \mathit{hash}).$$

   In what follows, an *rbc-validation for P_α in round k* is of the form

   $$(\textsf{rbc-validation}, k, \alpha, \mathit{hash}, \sigma),$$

   where σ is a valid S_{rbc}-signature on

   $$(\textsf{rbc-validation}, k, \alpha, \mathit{hash}).$$

   Party P_β will only do this once (i.e., if it sees a second authenticator from P_α in round k, it will ignore it), and it will also not do this if it has already executed Step 2.

2. If and when P_β receives

   - $n - 2t$ valid verifiable fragments corresponding to some authenticator

     $$(\textsf{authenticator}, k, \alpha, \mathit{hash}, \sigma),$$

     not necessarily the same authenticator it received in Step 1, and
   - either $n - t$ valid corresponding rbc-validation shares or a valid corresponding rbc-validation,

   party P_β will

   - reconstruct the corresponding block B,

- compute all of B's fragments and verify that *hash* is correct, and
- verify that B is a valid block (for this, it may need to wait until it receives a notarization for B's parent).

If the above verifications pass, P_β will construct the rbc-validation (if necessary), and broadcast

- the rbc-validation,
- B's authenticator and the notarization of B's parent (if $\alpha \neq \beta$).

Finally, P_β will *reliably receive* B (if $\alpha \neq \beta$).

Initial observations. One can verify that if the signature schemes are secure, the hash functions are collision resistant, and $t < n/3$, all of the essential properties of reliable broadcast are satisfied. Validity and integrity follow from the assumptions and the construction directly. For consistency, suppose some honest party P_β reliably receives a block B from a corrupt party P_α in round k. Because P_β has an rbc-validation, this means that at least $n - 2t$ honest parties broadcast corresponding fragments and rbc-validation shares. Since $t < n/3$, and an honest party only issues one rbc-validation share for P_α in round k, there can never be an rbc-validation for any other authenticator for P_α in round k. This implies consistency. It also implies totality, since at least $n - 2t$ parties already have broadcast corresponding fragments, and party P_β has broadcast all the remaining data that each other party will need to finish the protocol.

In fact, in our atomic broadcast protocol, we exploit a stronger property, which we call **strong totality**:

> *if $n - 2t$ honest parties have reliably received a block B from a party P_α in round k, then enough data has already been broadcast in order for any honest party to recover the block B.*

For the above protocol, if $n - 2t$ honest parties have reliably received a block B from a party P_α in round k, then some honest party besides P_α has reliably received B in Step 2, and by the arguments above, all the data needed has already been broadcast.

As we will see, in our atomic broadcast protocol, a party will only broadcast a notarization share on a block only if it has reliably received this block. As consequence of this, if any honest party has a notarized block B in its pool, then as soon as all messages currently in transit are delivered, all honest parties will be able to recover B as a notarized block (as well as all the ancestors of B in the blockchain ending at B).

Assume that the network is δ-synchronous.

If by time T, all honest parties have initiated a reliable broadcast instance from an honest party, then before time $T + 2\delta$, all honest parties will reliably receive that message.

Also, if by time T, all honest parties have initiated a reliable broadcast instance from a corrupt party, and some honest party Q reliably receives a message from that party, then before time $T + \delta$, all honest parties will reliably receive that message.

In terms of communication complexity, each party broadcasts at most one verifiable fragment per reliable broadcast instance. As we already saw, n such fragments have size at most $\approx 3S$, and so n^2 such fragments (which is the total number of fragments sent by all honest parties) have size at most $\approx 3Sn$. Importantly, not only is the overall communication complexity bounded, but it is well distributed over all the honest parties — in particular, the communication complexity contributed by the sender is only about twice that of the other parties.

5.3.3 A variant atomic broadcast protocol using reliable broadcast

We now present a variation on the protocol presented in Section 5.2, which we call Protocol ICC2, using the above reliable broadcast protocol as a subprotocol. The details are given in Figure 5. However, we elaborate on a few additional details here.

1. The sets $\mathcal{I}$ and $\mathcal{R}$ are initialized to $\emptyset$ at the start of each round.
    - A rank r is added to $\mathcal{I}$ by the reliable broadcast protocol when party P_α initiates a reliable broadcast instance from the party of rank r (either as sender or receiver).
    - A block B is added to $\mathcal{R}$ by the reliable broadcast protocol when party P_α *reliably receives a block from a party in round k.* In particular, when P_α proposes its own block B, the block B is immediately added to $\mathcal{R}$.
2. It is possible that a party P_α receives a notarization (or finalization) or a full set of notarization (or finalization) shares, even though it has not reliably received the corresponding block B, or even initiated a reliable broadcast instance from the corresponding party in the corresponding round. Here, we exploit the *strong totality* property discussed above. Indeed, such a notarization (or finalization) or set of notarization (or finalization) shares implies that at least $n - 2t$ honest parties reliably received the block, and hence P_α can eventually reconstruct the corresponding block B (and, by induction, all the ancestors of B in the blockchain ending at B).

    More concretely, in the "wait for" condition (a) in the Figure 5, party P_α will reconstruct the notarized block B from valid verifiable fragments it has received, if necessary, even if P_α did not reliably receive the block B or even initiate a corresponding reliable broadcast instance. Similar comments apply to finalized blocks in Figure 2.
3. In principle, once party P_α initiates a reliable broadcast instance from some party P_β in a given round k, it will carry it through to completion, regardless of whether or lower ranked block is later reliably received. However, once P_α has completed the round, it need not carry through any unfinished reliable broadcast instances for that round. This is justified by the strong totality property.

5.3.4 Analysis

Suppose the network is δ-synchronous, and the first honest party P enters the round at time T. Then before time $T + \delta$, all honest parties enter the round, including the leading honest party Q of rank h. This is justified by the strong totality property for the reliable broadcast protocol: if P ended the previous round with a notarized block B, all the data

```
broadcast a share of the round-1 random beacon

For each round $k = 1, 2, 3 \ldots$ :

    wait for $t + 1$ shares of the round-$k$ random beacon
    compute the round-$k$ random beacon (which defines the permutation $\pi$ for round $k$)
    broadcast a share of the random beacon for round $k + 1$

    let $r_{\text{me}}$ be the rank of $P_\alpha$ according to the permutation $\pi$
    $\mathcal{I} \leftarrow \emptyset$   // the set of ranks for which reliable broadcast has been initialized by $P_\alpha$
    $\mathcal{R} \leftarrow \emptyset$   // the set of blocks that have been reliably received by $P_\alpha$
    $\mathcal{N} \leftarrow \emptyset$   // the set of blocks for which notarization shares have been broadcast by $P_\alpha$

    $done \leftarrow \mathsf{false}$,  $t_0 \leftarrow \text{clock}()$

    repeat
        wait for either:
            (a)  a notarized round-$k$ block $B$,
                    or a full set of notarization shares for some valid
                    but non-notarized round-$k$ block $B$:
                 // Finish the round
                 combine the notarization shares into a notarization for $B$, if necessary
                 broadcast the notarization for $B$
                 $done \leftarrow \mathsf{true}$
                 if $\mathcal{N} \subseteq \{B\}$ then broadcast a finalization share for $B$

            (b)  a rank $r \in [r_{\text{me}}] \setminus \mathcal{I}$, such that
                    $\text{clock}() \geq t_0 + \Delta_{\text{prop}}(r)$, and
                    there is no block $B^* \in \mathcal{R}$ of rank $r^* \in [r]$:

                 initiate reliable broadcast for the party of rank $r$ in round $k$

            (c)  $r_{\text{me}} \notin \mathcal{I}$, $\text{clock}() \geq t_0 + \Delta_{\text{prop}}(r_{\text{me}})$, and $\mathcal{R} = \emptyset$:

                 // Propose a block
                 choose a notarized round-$(k-1)$ block $B_{\text{p}}$
                 $payload \leftarrow getPayload(B_{\text{p}})$
                 create a new round-$k$ block $B = (\mathsf{block}, k, \alpha, H(B_{\text{p}}), payload)$
                 initiate reliable broadcast from $P_\alpha$ in round $k$ of block $B$

            (d)  a block $B \in \mathcal{R} \setminus \mathcal{N}$ of rank $r$ such that
                    $\text{clock}() \geq t_0 + \Delta_{\text{ntry}}(r)$, and
                    there is no block $B^* \in \mathcal{R}$ of rank $r^* \in [r]$:

                 // Generate notarization share for B
                 $\mathcal{N} \leftarrow \mathcal{N} \cup \{B\}$, broadcast a notarization share for $B$
    until $done$
```

Figure 5: ICC2: Tree Building Subprotocol for party P_α

needed for all other honest parties to also see block B as a notarized block is already in flight at time T. As a consequence, before time $T + \delta + \Delta_{\text{prop}}(h)$, all honest parties are ready to initiate a broadcast from Q and all lower ranked parties.

Claim. Before time $T + 3\delta + \Delta_{\text{prop}}(h)$, there is some block B of rank $r \leq h$ that all honest parties have reliably received.

- On the one hand, suppose that by time $T + \delta + \Delta_{\text{prop}}(h)$, no honest party has reliably received any block of rank $< h$. This means that by that time, all honest parties will initiate reliable broadcast from Q, including Q itself, and less than 2δ time units later, all honest parties will reliably receive Q's proposal.

- On the other hand, suppose that by time $T + \delta + \Delta_{\text{prop}}(h)$, some honest party has reliably received any block of rank $< h$. Let B be the least ranked block that has been reliably received by any honest party at this time. It follows that all honest parties will reliably receive B less than δ time units later.

This claim allows us to guarantee liveness, provided $h = 0$ (i.e., the leader is honest) and

$$3\delta + \Delta_{\text{prop}}(0) \leq \Delta_{\text{ntry}}(1). \tag{23}$$

In the crash failure setting, liveness is guaranteed provided

$$3\delta + \Delta_{\text{prop}}(h) \leq \Delta_{\text{ntry}}(h+1). \tag{24}$$

If the delay functions are chosen linearly, as in (2), but with a slope of $3\Delta_{\text{bnd}}$, and the network remains Δ_{bnd}-synchronous, then the inequalities (23) and (24) will be satisfied. Importantly, just as in Section 4.8, parties may adaptively adjust the Δ_{ntry} function to compensate if Δ_{bnd} is too small, and thus maintain liveness even in that case.

We can also use this claim to bound the communication complexity of the protocol. Let $N_{\max}^{(k)}$ be the highest rank for which any honest party initiates a broadcast in round k. Then we have $N_{\max}^{(k)} < h + \ell$ provided

$$3\delta + \Delta_{\text{prop}}(h) \leq \Delta_{\text{prop}}(h + \ell). \tag{25}$$

If the delay functions are chosen linearly, as in (2), but with a slope of $3\Delta_{\text{bnd}}$, and the network does remain Δ_{bnd}-synchronous, we have bound $N_{\max}^{(k)} \leq h$, and we know $\mathrm{E}[h] = O(1)$ (see Section 4.3.1). Unlike for liveness, we cannot easily adaptively adjust the delay functions to compensate if Δ_{bnd} is too small.

As for latency, the bound corresponding to (4) becomes

$$\max(3\delta + \Delta_{\text{prop}}(h), \delta + \Delta_{\text{ntry}}(h)) + (h+1)\delta. \tag{26}$$

5.4 Restricting the set of block proposers

One can restrict the set of parties who propose blocks to those of ranks $0, \ldots, t$. All of the we have proved so far hold without change for this modification (combined as well with any of the variations given above). In fact, the bound in Lemma 2 on the number of notarized blocks per round drops from n to $t + 1$.

References

[AKK$^+$02] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. P. Stern. Scalable Secure Storage When Half the System Is Faulty. *Inf. Comput.*, 174(2):203–213, 2002.

[AKK$^+$07] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. P. Stern. Addendum to "Scalable secure storage when half the system is faulty" [Inform. Comput 174 (2)(2002) 203-213]. *Inf. Comput.*, 205(7):1114–1116, 2007.

[AMN$^+$20] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 106–118. IEEE, 2020.

[AMNR18] I. Abraham, D. Malkhi, K. Nayak, and L. Ren. Dfinity Consensus, Explored. Cryptology ePrint Archive, Report 2018/1153, 2018. `https://eprint.iacr.org/2018/1153`.

[ANRX21] I. Abraham, K. Nayak, L. Ren, and Z. Xiang. Good-case Latency of Byzantine Broadcast: A Complete Categorization, 2021. arXiv:2102.07240, `http://arxiv.org/abs/2102.07240`.

[BCG20] M. Bravo, G. Chockler, and A. Gotsman. Making Byzantine Consensus Live (Extended Version), 2020. arXiv:2008.04167, `http://arxiv.org/abs/2008.04167`.

[BDN18] D. Boneh, M. Drijvers, and G. Neven. Compact Multi-signatures for Smaller Blockchains. In T. Peyrin and S. D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464. Springer, 2018.

[Ben83] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In R. L. Probert, N. A. Lynch, and N. Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30. ACM, 1983.

[BKM18] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus, 2018. arXiv:1807.04938, `http://arxiv.org/abs/1807.04938`.

[BLS01] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. In C. Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[BR93] M. Bellare and P. Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 62–73. ACM, 1993.

[Bra87] G. Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

[CKS05] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.*, 18(3):219–246, 2005.

[CL99] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[CL02] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[CT05] C. Cachin and S. Tessaro. Asynchronous Verifiable Information Dispersal. In P. Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005.

[CWA$^+$09] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In J. Rexford and E. G. Sirer, editors, *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 153–168. USENIX Association, 2009. `http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf`.

[DFI20] DFINITY. A Technical Overview of the Internet Computer, 2020. `https://medium.com/dfinity/a-technical-overview-of-the-internet-computer-f57c62abc20f`.

[DGH$^+$87] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart, and D. B. Terry. Epidemic Algorithms for Replicated Database Maintenance. In F. B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 1–12. ACM, 1987.

[DLS88] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DRZ18] S. Duan, M. K. Reiter, and H. Zhang. BEAT: Asynchronous BFT Made Practical. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041. ACM, 2018.

[GAG$^+$19] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. 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, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 568–580. IEEE, 2019.

[GHM$^+$17] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. `https://eprint.iacr.org/2017/454`.

[GLT$^+$20] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Dumbo: Faster Asynchronous BFT Protocols. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 803–818. ACM, 2020.

[HMW18] T. Hanke, M. Movahedi, and D. Williams. DFINITY Technology Overview Series, Consensus System, 2018. arXiv:1805.04548, `http://arxiv.org/abs/1805.04548`.

[Lib20] LibraBFT Team. State Machine Replication in the Libra Blockchain, 2020. `https://diem-developers-components.netlify.app/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2020-05-26.pdf`.

[LSP82] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Mer80] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 14-16, 1980*, pages 122–134. IEEE Computer Society, 1980.

[Mer89] R. C. Merkle. A Certified Digital Signature. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[MXC$^+$16] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT Protocols. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference* *on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

[NBMS19] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman. Gogsworth: Byzantine View Synchronization, 2019. arXiv:1909.05204, `http://arxiv.org/abs/1909.05204`.

[NK20] O. Naor and I. Keidar. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR, 2020. arXiv:2002.07539, `http://arxiv.org/abs/2002.07539`.

[PS18] R. Pass and E. Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018.

[PSL80] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

[RC05] H. V. Ramasamy and C. Cachin. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. In J. H. Anderson, G. Prencipe, and R. Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2005.

[Riz97] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *Comput. Commun. Rev.*, 27(2):24–36, 1997.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[SDV19] C. Stathakopoulou, T. David, and M. Vukolic. Mir-BFT: High-Throughput BFT for Blockchains, 2019. arXiv:1906.05552, `http://arxiv.org/abs/1906.05552`.

[Sha79] A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.