Internet Computer Consensus

Extended Abstract

Jan Camenisch, Manu Drijvers, Timo Hanke,
Yvonne-Anne Pignolet, Victor Shoup, Dominic Williams
firstname.surname@dfinity.org
DFINITY Foundation
Zurich, Switzerland

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 simple and robust: in any round where the leader is corrupt (which itself happens with probability less than 1/3) or the network is asynchronous, each ICC protocol will effectively allow other parties to step in and propose blocks for that round and to move the protocol forward to the next round. In case there was no agreement on a single block in a round, a decision for this round will be taken in a later round with synchronous network behavior and an honest leader. The task of reliably disseminating the blocks to all parties is an integral part the protocol. An additional property enjoyed by the ICC protocols 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). The first of these protocols (ICC0) illustrates the combination of the main building blocks in a simplified manner for an easier presentation and analysis. Protocol 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 must address. 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.

CCS CONCEPTS

• **Theory of computation → Computational complexity and cryptography**; **Design and analysis of algorithms**; **Distributed algorithms**.

KEYWORDS

consensus, blockchain, internet computer, atomic broadcast

ACM Reference Format:
Jan Camenisch, Manu Drijvers, Timo Hanke,, Yvonne-Anne Pignolet, Victor Shoup, Dominic Williams. 2022. Internet Computer Consensus: Extended Abstract. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC '22), July 25–29, 2022, Salerno, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3519270.3538430

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* [33]: 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* [9, 16, 33].

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 [18]. 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* in the full version of this paper (they do not rely on unspecified, non-standard components), *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 in our analysis.

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 [20]. 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 (described in more detail later).

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. As a consequence, in contrast to protocols which only switch leaders if they misbehave, no additional timeouts and/or protocol logic is necessary to determine if a leader is honest. This keeps the protocol easy to implement and analyse.

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.

The *round complexity* for the ICC protocols can be defined as the number of rounds until a block is committed in the worst case. For a static adversary, this complexity is $O(1)$ for the ICC protocols in expectation and $O(\log n)$ with high probability. If the adversary is adaptive (not analysed formally in this paper) and hence can corrupt nodes according to the random beacon used in a particular round, the worst case message complexity per round stays the same, but the round complexity increases to $O(n)$. Note that, regardless of the amount of time passed until a block is committed, the recursive nature of the ICC protocols ensures that eventually one block will be committed for every round.

Of course, message and round complexity do 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 [17]. 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 sub-protocol 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* [30], 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 [25]. The first solution in the synchronous communication model was given by [31].

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 [4], 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 [9, 10], with significant improvements more recently in [2, 19, 23, 27]. For example, [2] reaches agreement in $O(1)$ rounds and exchanging $O(n^2)$ messages in expectation, even against an adaptive adversary.

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 [20]. The first truly practical protocol in this setting is the well-known PBFT protocol [13, 14], 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. Thus the round complexity of PBFT is $O(1)$.

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

(1) 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 described, 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.

(2) 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 [22, 32, 36]. However, [35] provides an empirical study suggesting that 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 [35], the network was a global wide-area network, which is the setting of most interest to us in this work. As was reported in [35], 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, [35] argues that approaches such as those in [22, 32, 36] only exacerbate the bottleneck at the leader.

There has also been recent work on replacing the view-change subprotocol of PBFT with rapid leader rotation, for example HotStuff [36] and Tendermint [8]. Like PBFT, both of these protocols are leader based; however, they do not rely on a view-change subprotocol, and in fact may change the 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) and its round complexity is $O(1)$ under a static adversary. 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. Hotstuff requires a linear number of messages when the network is synchronous and the leader is honest. It features a worst case message complexity of $O(n^2)$ when the network is synchronous. In comparison, ICC's worst case message complexity is $O(n^3)$ and in synchronous rounds it is $O(n^2)$ with overwhelming probability. When considering a *weak* adaptive adversary, which requires more than one round to corrupt nodes, then the adversary cannot compromise the ICC leader of the next round fast enough. In contrast, if Hotstuff uses a fixed leader rotation setup, it is susceptible to such a weak adaptive adversary causing $O(n)$ leader changes. As mentioned earlier, [2] reaches agreement in $O(1)$ rounds and exchanging $O(n^2)$ messages in expectation, even against an adaptive adversary and under asynchrony.

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. The task of the pacemaker subprotocol is less onerous than that of the view-change subprotocol. LibraBFT (a.k.a., DiemBFT) [26] 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 [7, 28, 29]. 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 can also achieve a worst case message complexity of $O(n^2)$ when the network is synchronous. It 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. 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 Δ_{bnd} that may be significantly larger than the actual network delay δ, and in Tendermint, every round takes time $O(\Delta_{\text{bnd}})$, even when the leader is honest.

MirBFT [35] 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 view-change subprotocol — however, as pointed out in [35], 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 [35].

Algorand [21] 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 simple and entirely self contained.
- 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 [11], and previously used in the context of atomic broadcast in [27]. We propose a new erasure-coded reliable broadcast subprotocol with better latency than that in [11], 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 [15], 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, [15] show that the throughput of existing implementations of PBFT drops to zero under certain types of (quite simple) Byzantine behavior. The paper [15] 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 [15]: in any round where the leader is corrupt (which itself happens with probability less than 1/3), each ICC protocol will effectively allow other parties to step in and propose blocks for that round and 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 $O(\Delta_{\text{bnd}})$, where $\Delta_{\text{bnd}} \geq \delta$ is the network-delay bound on which the partial synchrony assumption (used to ensure liveness) is based. In case there was no agreement on a single block in a round, a decision for this round will be taken in a later round with synchronous network behavior and an honest leader.

Preliminary versions of the ICC protocols. Note that the protocols presented here are very different from those discussed in either [24] or [1]. In particular, unlike the protocols presented here, the preliminary protocols in [1, 24] (1) only guaranteed safety in a synchronous setting, (2) were not optimistically responsive, and (3) had potentially unbounded communication complexity.

Roadmap. Section 2 defines the cryptographic primitives the ICC protocols rely on. In Section 3, we describe the ICC0 in detail and state its main properties formally. In Section 5 we present performance numbers from the Internet Computer deployment. All proofs and detailed descriptions and analysis of ICC1 and ICC2 are deferred to the full version [12] due to space constraints.

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.

Such threshold signatures 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 [5], in which a signature share is an ordinary BLS signature [6], 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 [34]) 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 $R_0, R_1, R_2 \dots$. The value R_0 is a fixed, initial value, known to all parties. For $k = 1, 2, \dots$, 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" [3]).

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 $\{0, \dots, k - 1\}$.

We have n parties, $P_1, \dots, P_n$. 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 full version [12]).

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 generally assume that the 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 [13]. 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 B, which consists of

- a round number (which is also the depth of B 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

$$(\text{block}, k, \alpha, \textit{phash}, \textit{payload}). \tag{1}$$

[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.

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 $phash = H(B_{\text{p}})$ 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 $(\mathsf{notarization\text{-}share}, k, \alpha, H(B), ns, \beta)$, where ns is a valid S_{notary}-signature share on $(\mathsf{notarization}, k, \alpha, H(B))$ by party P_β.
- 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 $(\mathsf{finalization\text{-}share}, k, \alpha, H(B), fs, \beta)$, 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 root $= B_0, B_1, \ldots, B_k = B$ in its pool that form a **blockchain**, meaning that B_i is B_{i+1}'s parent, for $i = 0, \ldots, k-1$, along with authenticators for $B_1, \ldots, B_k$ and notarizations for $B_1, \ldots, B_{k-1}$.

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:

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

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).

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 $\text{rank}_\pi(B)$ 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 $\Delta_{\text{prop}}(r_{\text{me}})$ time units have elapsed since the beginning of the round (more precisely, since the time at which it executes the step $t_0 \leftarrow \text{clock}()$ 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

```
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 $\pi$
    $\mathcal{N} \leftarrow \emptyset$   // 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 ← false
    proposed ← 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 ← true
                if $\mathcal{N} \subseteq \{B\}$ then broadcast a finalization share for $B$

            (b) not proposed and clock() $\geq t_0 + \Delta_{\text{prop}}(r_{\text{me}})$:
                // Propose a block
                choose a notarized round-$(k-1)$ block $B_{\text{p}}$
                payload ← getPayload($B_{\text{p}}$)
                create a new round-$k$ block $B = (\text{block}, k, \alpha, H(B_{\text{p}}), payload)$
                broadcast $B$, $B$'s authenticator, and the notarization for $B$'s parent
                proposed ← true
            (c) a valid round-$k$ block $B$ of rank $r$ such that $B \notin \mathcal{N}$, $r \notin \mathcal{D}$, clock() $\geq t_0 + \Delta_{\text{ntry}}(r)$, 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 $r \neq r_{\text{me}}$ 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

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 $B_{\text{p}} = \text{root}$). 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 $getPayload(B_{\text{p}})$, 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 $\Delta_{\text{ntry}}(t)$ 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;
- 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.

```
$k_{\max} \leftarrow 0$ // 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

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 PROPERTIES OF PROTOCOL ICC0

We now state the main properties of Protocol ICC0, corresponding to Properties P1, P2, and P3 discussed in Section 3.3. The following lemma corresponds to Property P1.

Lemma (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.*

The following lemma corresponds to Property P2.

Lemma (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.*

The following lemma corresponds to Property P3. For this, we formally state our partial synchrony assumption:

Definition. *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 $T + \delta$.*

Lemma (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.

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

These lemmas are proved in the full version of this paper [12]. In addition, the full version contains an analysis of the *expected* message complexity, latency, and various other performance metrics. Furthermore, we also present and analyse Protocols ICC1 and ICC2 in detail in that version.

5 PERFORMANCE MEASUREMENTS

The Internet Computer currently consists of 518 nodes running in 33 independent data centers worldwide.[2] It is partitioned into 35 shards (called *subnets*), each of them running its own instance of consensus with 13 to 40 nodes of which at most three are located in the same data center. The code is open source.[3]

We measured all outgoing traffic per node over a 5min window for three scenarios with a small and a large subnet, see Table 1 . The observed ping RTT between nodes in different data centers varies between 6ms and 110ms with a packet loss probability below 0.001.

In the first scenario, no payload from users is included in the blocks, i.e., they only contain management information. The current parametrization leads to 1.1 blocks per second on small subnets and about 0.4 blocks per second on large subnets. Note that the traffic includes more than only the messages for consensus, e.g., messages exchanged with the clients, the periodic cryptographic key resharing scheme, logs, metrics etc. In the second scenario, the subnets are each exposed to 100 state changing requests per second, each of them carrying 1KB of user payload. Higher user throughput is possible, these numbers have simply been picked to illustrate the overhead introduced by the protocol. In the third scenario, one third of the nodes refuses to participate in the protocol leading to a lower average number of blocks finalized per second and a decrease of the traffic sent per node.

	without load	**with load**	**with load and node failures**
13 node subnet	1.09 blocks/s 1.64 Mb/s	1.10 blocks/s 4.72 Mb/s	0.45 blocks/s 4.39 Mb/s
40 node subnet	0.41 blocks/s 4.63 Mb/s	0.41 blocks/s 7.32 Mb/s	0.16 blocks/s 5.06 Mb/s

Table 1: Average block rate and sent traffic.

[2]https://dashboard.internetcomputer.org/
[3]https://github.com/dfinity/ic

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2018. Dfinity Consensus, Explored. Cryptology ePrint Archive, Report 2018/1153. https://eprint.iacr.org/2018/1153.

[2] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 337–346.

[3] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby (Eds.). ACM, 62–73. https://doi.org/10.1145/168588.168596

[4] Michael Ben-Or. 1983. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, Robert L. Probert, Nancy A. Lynch, and Nicola Santoro (Eds.). ACM, 27–30. https://doi.org/10.1145/800221.806707

[5] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact Multi-signatures for Smaller Blockchains. In *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 (Lecture Notes in Computer Science, Vol. 11273)*, Thomas Peyrin and Steven D. Galbraith (Eds.). Springer, 435–464. https://doi.org/10.1007/978-3-030-03329-3_15

[6] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short Signatures from the Weil Pairing. In *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 (Lecture Notes in Computer Science, Vol. 2248)*, Colin Boyd (Ed.). Springer, 514–532. https://doi.org/10.1007/3-540-45682-1_30

[7] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2020. Making Byzantine Consensus Live (Extended Version). arXiv:2008.04167, http://arxiv.org/abs/2008.04167.

[8] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. arXiv:1807.04938, http://arxiv.org/abs/1807.04938.

[9] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2139)*, Joe Kilian (Ed.). Springer, 524–541. https://doi.org/10.1007/3-540-44647-8_31

[10] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.* 18, 3 (2005), 219–246. https://doi.org/10.1007/s00145-005-0318-0

[11] Christian Cachin and Stefano Tessaro. 2005. Asynchronous Verifiable Information Dispersal. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3724)*, Pierre Fraigniaud (Ed.). Springer, 503–504. https://doi.org/10.1007/11561927_42

[12] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. 2021. Internet Computer Consensus. Cryptology ePrint Archive, Report 2021/632. https://ia.cr/2021/632.

[13] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, Margo I. Seltzer and Paul J. Leach (Eds.). USENIX Association, 173–186. https://dl.acm.org/citation.cfm?id=296824

[14] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[15] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 153–168. http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf.

[16] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 36, 4 (2004), 372–421.

[17] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, Fred B. Schneider (Ed.). ACM, 1–12. https://doi.org/10.1145/41840.41841

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

[19] Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2028–2041. https://doi.org/10.1145/3243734.3243812

[20] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. Cryptology ePrint Archive, Report 2017/454. https://eprint.iacr.org/2017/454.

[22] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. 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*. IEEE, 568–580. https://doi.org/10.1109/DSN.2019.00063

[23] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster Asynchronous BFT Protocols. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 803–818. https://doi.org/10.1145/3372297.3417262

[24] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. DFINITY Technology Overview Series, Consensus System. arXiv:1805.04548, http://arxiv.org/abs/1805.04548.

[25] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

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

[27] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 31–42. https://doi.org/10.1145/2976749.2978399

[28] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2019. Gogsworth: Byzantine View Synchronization. arXiv:1909.05204, http://arxiv.org/abs/1909.05204.

[29] Oded Naor and Idit Keidar. 2020. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. arXiv:2002.07539, http://arxiv.org/abs/2002.07539.

[30] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with Optimistic Instant Confirmation. In *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 (Lecture Notes in Computer Science, Vol. 10821)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, 3–33. https://doi.org/10.1007/978-3-319-78375-8_1

[31] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (1980), 228–234. https://doi.org/10.1145/322186.322188

[32] HariGovind V. Ramasamy and Christian Cachin. 2005. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3974)*, James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer (Eds.). Springer, 88–102. https://doi.org/10.1007/11795490_9

[33] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167

[34] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[35] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. arXiv:1906.05552, http://arxiv.org/abs/1906.05552.

[36] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT Consensus in the Lens of Blockchain. arXiv:1803.05069, http://arxiv.org/abs/1803.05069.