Information Theoretic HotStuff

Ittai Abraham
VMWare Research, Herzliya, Israel

Gilad Stern
The Hebrew University in Jerusalem, Israel

Abstract

This work presents Information Theoretic HotStuff (IT-HS), a new optimally resilient protocol for solving Byzantine Agreement in partial synchrony with information theoretic security guarantees. In particular, IT-HS does not depend on any PKI or common setup assumptions and is resilient to computationally unbounded adversaries. IT-HS is based on the Primary-Backup view-based paradigm. In IT-HS, in each view, and in each view change, each party sends only a constant number of words to every other party. This yields an $O(n^2)$ word and message complexity in each view. In addition, IT-HS requires just $O(1)$ persistent local storage and $O(n)$ transient local storage. Finally, like all Primary-Backup view-based protocols in partial synchrony, after the system becomes synchronous, all nonfaulty parties decide on a value in the first view a nonfaulty leader is chosen. Moreover, like PBFT and HotStuff, IT-HS is optimistically responsive: with a nonfaulty leader, parties decide as quickly as the network allows them to do so, without regard for the known upper bound on network delay. Our work improves in multiple dimensions upon the information theoretic version of PBFT presented by Miguel Castro, and can be seen as an information theoretic variant of the HotStuff paradigm.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases byzantine agreement, partial synchrony, bounded space

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.11

Related Version A full version of the paper is available at `https://arxiv.org/abs/2009.12828`.

Funding *Gilad Stern*: This work was supported by the HUJI Federmann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate (INCD) in the Prime Minister's Office.

1 Introduction

This work assumes the model of Castro and Liskov's PBFT protocol [7, 9, 11]. In particular we deal with the task of Byzantine Agreement in a partially synchronous network. The setting of partial synchrony was proposed by Dwork, Lynch, and Stockmeyer [12] and studied extensively since. In this model, the network starts off as an asynchronous network and at some unknown time becomes synchronous with a known delay Δ on message arrival. This time is known as the Global Stabilization Time, or GST in short. This model turns out to be a useful one, managing to capture some of the behaviour of real-world networks. As in PBFT, our goal in this work is to reduce the use of cryptographic tools that require a computationally bounded adversary as much as possible. Much like PBFT, our algorithm is *information theoretically secure*. Formally, as in PBFT [7, 9, 11], our protocol is secure against adversaries that are not computationally bounded under the assumption that there exist authenticated channels that can be made secure against such adversaries. For example, authenticated channels can be obtained via a setup of one time pads or via Quantum key exchange [2].

There are several good reasons to design protocols in the information theoretic security setting. First, from a theoretical perspective we are interested in minimizing the assumptions. Fewer assumptions often tend to add clarity and conceptual simplicity. Secondly, adding public-key cryptography primitives adds a performance overhead and increases the code-base attack surface, whereas computations in the information-theoretic setting are quick and often amount to simple memory management and counting. Finally, protocols in this setting are more "future-proof". Such protocol are more resilient to breaking certain cryptographic assumptions and to major technological disruptions in the field.

The PBFT variants that use a PKI and digital signatures can easily use bounded storage at each party (per active slot). One of the challenges of the PBFT protocol when only authenticated channels (no signatures) are used is that obtaining bounded storage is not immediate. Indeed all the peer reviewed papers that we are aware of obtain unbounded solutions [7, 10]. Castro's thesis [9] does include a bounded storage solution, however to the best of our knowledge this result was not published in a peer reviewed venue, and its complexity does rely on cryptographic hash functions.

1.1 Main result

Our main result is *Information Theoretic HotStuff* (IT-HS), a protocol solving the task of Byzantine Agreement in partial synchrony with information theoretic security using bounded storage that sends messages whose maximal size is $O(1)$ words (both during a view and during a view change). The protocol is resilient to any number of Byzantine parties f such that $n > 3f$, making it optimally resilient. In the protocol, there are several virtual rounds called views, and each one has a leader, called a primary. This is a common paradigm for solving Byzantine agreement, famously used in the Paxos protocol [16] and in later iterations on those ideas such as PBFT [7, 9, 15] and more recent protocols in the Blockchain era [4, 5, 6, 14, 18]. We use a standard measure of storage called a *word* and assume a word can contain enough information to store any command, identifier, or counter. Formally, this means that much like in all previous systems and protocols, our counters, identifiers, and views are bounded (by say 256 bits). In IT-HS, in each view and in each view change, each party sends just a constant number of words and messages to each other party, making the total word and message complexity $O(n^2)$ in each view and in each view change. As far as we know, this is the best known communication complexity and word complexity for information theoretic protocols of this kind (see table below for comparison). In addition, all parties require $O(n)$ space throughout the protocol, out of which only $O(1)$ space needs to be persistent, crash-resistant memory. Clearly at least $O(1)$ persistent memory is required, because otherwise a decided upon value can be "forgotten" by all parties if they crash and reboot. As far as we know, $O(n)$ transient space complexity is the best known result. In the shared memory model, a lower bound of $\Omega(n)$ registers exists [13], suggesting that the total amount of persistent memory in the system is optimal.

In IT-HS, all nonfaulty parties are guaranteed to decide on a value and terminate during the first view a nonfaulty party is chosen as primary after GST, if they haven't done so earlier. This is the asymptotically optimal convergence for such protocols: For deterministic leader rotation this implies $O(f)$ rounds after GST. If we assume that parties have access to a randomized leader-election beacon, then this implies $O(1)$ expected rounds after GST. Furthermore, like PBFT and HotStuff, IT-HS is *optimistically responsive.* If the network delay is actually $\delta = o(\Delta)$, all nonfaulty parties terminate in $O(\delta)$ time instead of in $O(\Delta)$ time. IT-HS uses an *asymptotically* optimal (constant) number of rounds given a nonfaulty primary and after the network becomes synchronous.

The most relevant related works for IT-HS are the PBFT protocol variants [7, 9, 10, 11] and the HotStuff protocol variants [18]. The following table provides a comparison between them.

	Assumptions	Persistent storage	Maximum size of message (in words)
PBFT (OSDI) [11]	PKI	$\Omega(n)$	$O(n)$
PBFT (TOCS) [10]	Authenticated Channels, Cryptographic Hash	$\Omega(n)$ per view (unbounded)	$\Omega(n)$ per view (unbounded)
PBFT (Thesis) [9]	Authenticated Channels, Cryptographic Hash	$O(1)$	$O(n)$
YAVP (Cachin) [7]	Authenticated Channels, Cryptographic Hash	$\Omega(n)$ per view (unbounded)	$\Omega(n)$ per view (unbounded)
HotStuff (authenticators) [18]	PKI	$O(n)$	$O(n)$
HotStuff (threshold sig) [18]	DKG: Threshold signature setup	$O(1)$	$O(1)$ (threshold sig)
IT-HS (this work)	Authenticated Channels	$O(1)$	$O(1)$

As mentioned earlier, all previous peer-reviewed works in the information theoretic setting require at least $\Omega(n \cdot v)$ words of storage, where v is the view number. Since the view number can grow arbitrarily large, the persistent storage requirement is unbounded. The only work we know of that achieves comparable asymptotic performance relies on the relatively strong cryptographic assumption of threshold signatures.

We note that IT-HS does not only use fewer assumptions (does not use any cryptographic hash function), it also obtains the asymptotically optimal $O(1)$ word bound on the maximal message size. All other protocols require at least $\Omega(n)$ size messages to be sent during view change by the primary (except for Hotstuff when using a Distributed Key Generation setup and threshold signatures).

Compared to PBFT, our work can be seen as addressing the open problem left in the PBFT journal version (which uses unbounded space and cryptographic hash functions) and is an improvement of the non peer-reviewed PBFT thesis work (which still uses cryptographic hash functions). IT-HS obtains the same $O(1)$ persistent space, and manages to reduce the maximum message size from $O(n)$ (in the PBFT view change) to the asymptotically optimal $O(1)$ maximum message size and requires no cryptographic hash functions.

Relative to HotStuff, our work shows that without any PKI (public key infrastructure) or DKG (distributed key generation) assumptions and without any cryptographic setup ceremony, constant size messages and constant size persistent storage are possible! We do note that IT-HS requires $O(n^2)$ messages and words per view, while the Hostuff version with a DKG setup that uses threshold signatures requires just $O(n)$ messages and words per view. On the other hand, HS-IT requires no cryptographic setup ceremony and no computational assumptions other than pairwise authenticated channels. Like HS-IT, all other protocols that do not use threshold signatures (even those that require a PKI) use $\Omega(n^2)$ words per view.

Our contributions

1. Unlike previous solutions which used cryptographic hash functions and required $O(n)$ sized messages, We provide the first information theoretic primary backup protocol where all messages have size $O(1)$ and storage is bounded to size $O(1)$.
2. We manage to reduce the size of the view change messages to a constant by adapting the HosStuff paradigm without using any cryptographic primitives. We introduce an information theoretic technique for one-transferable signatures to maintain bounded space and adopt the view change protocol accordingly.

3. Without using any cryptographic primitives, we obtain a protocol that requires just a constant amount of persistent storage. We use information theoretic techniques that require storing just the last two events from each message type.

1.2 Main Techniques

As the name might suggest, IT-HS is inspired by the Tendermint, Casper, and HotStuff protocols [4, 5, 6, 18] and adapts them to the information theoretic setting. We show how to adapt the *lock and key* mechanism which was suggested in HotStuff [18] and made explicit in [1], to the information theoretic setting while maintaining just $O(1)$ persistent storage. In a basic *locking* mechanism [4, 12], before nonfaulty parties decide on a value, they set a "lock" that doesn't allow them to respond to primaries suggesting values from older views. Then, before deciding on a value, nonfaulty parties require a proof that enough parties are locked on the current view. This ensures that if some value is decided upon, there will be a large number of nonfaulty parties that won't be willing to receive messages from older views, and thus this will remain the only viable value in the system.

The challenge with the locking mechanism is that the adversary can cause nonfaulty locked parties to block nonfaulty primaries, unless the primary waits for all nonfaulty parties to respond. To overcome this, an additional round is added so that a nonfaulty locked party guarantees that there is a sufficient number of nonfaulty parties with a *key*. When a new primary is chosen, it waits for just $n - f$ parties to send their highest keys, and uses the highest one it receives.

The challenge with using a key is verifying its authenticity. In the cryptographic setting, this is easily done using signatures. In the information theoretic setting, verification is more challenging. One approach is using Bracha's Broadcast [3] in order to prove that the key received by the primary will also be accepted by the other parties. Since there is no indication of termination in Bracha's Broadcast, there is a need to maintain an unbounded number of broadcast instances (one for each view). Using such techniques requires an unbounded amount of space.

To overcome this challenge with bounded space, we propose a novel approach of using *one-hop transferable* proofs. If before moving to the next round, a nonfaulty party hears from $n - f$ parties, then it knows it heard from at least $f + 1$ nonfaulty parties. This means that once the system becomes synchronous, every party will hear from those $f + 1$ parties and know that at least one of them is nonfaulty. We use this type of "one-hop transferable proof" twice so we have 3 key messages instead of one, each proving that the next key (or lock) is correct, and that this fact can be proven to other parties, thereby ensuring liveness.

In order to send just a constant number of words, we send just the last two times that the value of the *key* was updated. If the final update to *key* happened after a lock was set, and its value is different than the lock's value, then the lock is safe to open. Otherwise, if the older of the two updates was after the lock's view then at least in one of those times it was updated to a value other than the lock's value, and thus the lock is also safe to open. Using this idea, parties can also prove to a primary that a $key3$ suggestion is safe. In this case, the parties either show a later view in which the same value was set for $key2$, or two later views in which the value of $key2$ was updated. This proof shows that any previous lock either has the same value as $key3$, or can be opened safely regardless of its value. The idea of storing just two lock values appears in Castro's Thesis [9], we significantly extend this technique to use our novel *one-hop transferable* information theoretic "signatures" combined with the HotStuff keys-lock approach.

1.3 Protocol Overview

Much like all primary backup protocols, each view of IT-HS consists of a constant number of rounds. Each party waits to receive $n - f$ round i messages before it sends a round $i + 1$ message (in some rounds there are additional checks). Much like PBFT, each round involves an all-to-all message sending format. Throughout the protocol, parties may set a lock for a given view and value. This lock indicates that any proposal for a different $view, value$ pair should not be accepted without ample proof that another value reached advanced stages in a later view. In order to provide that proof, the parties send a $proof$ message that helps convince parties with locks to accept messages about a different value if appropriate.

The rounds of IT-HS for a given view can be partitioned into 4 parts:

1. View Change: parties first send a *request* message, indicating that they started the view. Once parties hear the *request* message sent by the primary, they respond with their current suggestion for a value to propose, as well as the view in which this suggestion originated, and additional data which will help validate all nonfaulty parties' suggestions (proofs). After receiving those suggestions, the primary checks whether each suggestion is valid, and once it sees $n - f$ valid suggestions, it sends a *propose* message for the one that originated in the most recent view.
2. Propose message round: this is where a party checks a proposal relative to its lock. Each party checks if it's locked on the same value as the one proposed, or convinced to override its lock by $f + 1$ proof messages. If that is the case, it responds by sending an *echo* message.
3. Key message rounds: this is where a key is created that can be later used to unlock parties. After receiving $n - f$ *echo* messages with the same value, parties send a $key1$ message with that value. After receiving $n - f$ $key1$ messages with the same value they send a $key2$ message. After receiving $n - f$ $key2$ messages with the same value they send a $key3$ message. We use these three rounds in order to obtain transferable information theoretic signatures on the key message.
4. Lock and commit rounds: After receiving $n - f$ $key3$ messages with the same value they *lock* on it and send a *lock* message. After receiving $n - f$ *lock* messages with the same value they *commit* and send a *done* message.

Before sending a $key1$ message, the local $key1$, $key1\_val$ and $prev\_key1$ fields are updated. These fields contain the last view in which a $key1$ message was sent, its value, and the last view a $key1$ message was sent with a different value. Similar updates take place for the other *key* fields and the *lock* fields. The *echo*, *lock* and various *key* messages are tagged with the current view, while the *done* message is a protocol-wide message and isn't related to a specific view. Similarly to the mechanism in Bracha Broadcast [3], after receiving $f + 1$ *done* messages, the message is echoed, and after receiving $n - f$ messages it is accepted and the parties decide and terminate. If a party sees that this view takes more than the expected time, it sends an *abort* message for the view. The same $f + 1$ threshold for echoing the *abort* message and $n - f$ threshold for moving to the next view are implemented in order to achieve the same properties. In order to avoid buffering *request* and *abort* messages, only the messages with the highest view v are actually stored and are understood as a *request* or *abort* message for any view up to v.

2 Byzantine Agreement in Partial Synchrony

This section deals with the task of Byzantine Agreement in a partially synchronous system. In this model, there exist n parties who have local clocks and authenticated point-to-point channels to every other party. The system starts off fully asynchronous: the clocks are not

Algorithm 1 IT-HS.

```
Code for party $i$ with input $x_i$:
1: $lock \leftarrow 0, lock\_val \leftarrow x_i$
2: $key3 \leftarrow 0,\ key3\_val \leftarrow x_i$
3: $key2 \leftarrow 0, key2\_val \leftarrow x_i, prev\_key2 \leftarrow -1$
4: $key1 \leftarrow 0, key1\_val \leftarrow x_i, prev\_key1 \leftarrow -1$
5: $view \leftarrow 0$
6: $\forall j \in [n]\ \ highest\_request\,[j] \leftarrow 0$
7: continually run $check\_progress()$ in the background
8: while true do  ▷ memory from last $process\_messages$ and $view\_change$ calls is freed
9:     $cur\_view \leftarrow view$
10:    as long as $cur\_view = view$, run
11:        at time $cur\_time() + 11\Delta$ do
12:            send an $\langle abort, view\rangle$ message to all parties
13:        ignore messages from other views, other than $abort$, $done$ and $request$ messages
14:        $primary \leftarrow (view \mod n) + 1$
15:        continually run $process\_messages(view)$ in the background
16:        $view\_change(view, primary)$
```

synchronized, and every message can be delayed any finite amount of time before reaching its recipient. At some point in time, the system becomes fully synchronous: the clocks become synchronized, and every message (including the ones previously sent) arrives in Δ time at most, for some commonly known Δ. It is important to note that even though it is guaranteed that the system eventually becomes synchronous, the parties do not know when it is going to happen, or even if it has already happened. The point in time in which the system becomes synchronous is called the Global Stabilization Time, or GST in short. In the setting of a Byzantine adversary, the adversary can control up to f parties, making them arbitrarily deviate from the protocol. In general, throughout this work assume that $f < \frac{n}{3}$.

▶ **Definition 1.** *A Byzantine Agreement protocol in partial synchrony has the following properties:*

- ***Termination.*** *If all nonfaulty parties participate in the protocol, they all eventually decide on a value and terminate.*
- ***Correctness.*** *If two nonfaulty parties decide on values val, val', then $val = val'$.*
- ***Validity.*** *If all parties are nonfaulty and they all have the same input val, then every nonfaulty party that decides on a value does so with the value val.*

We note that if we assume the parties have access to an external validity function, as described in [8], this protocol can be easily adjusted to have external validity. In this setting, the external validity function defines which values are "valid", and all nonfaulty parties are required to output a valid value. The only adjustment needed is for parties to also check if a value is valid before sending an *echo* message.

The main goal of this section is to show that Algorithm 1 is a Byzantine Agreement protocol in partial synchrony resilient to $f < \frac{n}{3}$ Byzantine parties. For ease of discussion, a party is said to perform an action "in view v" if when it performed the action its local *view* variable equaled v. In addition, we define the notion of messages "supporting" a key or opening a lock:

Algorithm 2 view_change(view,primary).

```
Code for party $i$:
1: send $\langle request, view\rangle$ to all parties $j \in [n]$
2: upon $highet\_request\,[primary] = view$, do
3:     send $\langle suggest, key3, key3\_val, key2, key2\_val, prev\_key2, view\rangle$ to $primary$
4: $send\_all\_upon\_join(\langle proof, key1, key1\_val, prev\_key1, view\rangle)$
5: if $primary = i$ then
6:     $suggestions \leftarrow \emptyset$
7:     $key2\_proofs \leftarrow \emptyset$
8:     upon receiving the first $\langle suggest, k3, v3, k2, v2, pk2, view\rangle$ message from $j$, do
9:         if $pk2 < k2 < view$ then
10:            add $(k2, v2, pk2)$ to $key2\_proofs$
11:        if $k3 = 0$ then
12:            add $(k3, v3)$ to $suggestions$
13:        else if $k3 < view$ then
14:            upon $accept\_key\,(k3, v3, key2\_proofs) = true$, do
15:                add $(k3, v3)$ to $suggestions$
16:    wait until $|suggestions| \geq n - f$, then do
17:        let $(k, v) \in suggestions$ be some tuple such that $\forall\,(k', v') \in suggestions\ k' \leq k$
18:        $send\_all\_upon\_join(\langle propose, k, v, view\rangle)$
```

▶ **Definition 2.** *A suggest message is said to support the pair $key3, key3\_val$, if its $key2$, $key2\_val$, and $prev\_key2$ fields are ones for which at least one of the conditions in the loop of Algorithm 3 is true.*

A proof message is said to support opening the pair $lock, lock\_val$ if its $key1$, $key1\_val$, and $prev\_key1$ fields are ones for which at least one of the conditions in the loop of Algorithm 7 is true.

Before proving that Algorithm 1 is a Byzantine Agreement protocol in partial synchrony, we prove several lemmas. The lemmas can be classified into two types: safety lemmas and liveness lemmas. The safety lemmas show that if a nonfaulty party decides on some value, no nonfaulty party decides on a different value. This is achieved by the locking mechanism. Roughly speaking, if some nonfaulty party decides on some value, there exist $f+1$ nonfaulty parties that are locked on that value and will stop any other value from progressing past the *propose* message. The liveness lemmas show two crucial properties for liveness. First of all, if some nonfaulty party sets $key3$ to be some value, then there are $f+1$ parties that will support that key. This means that if a nonfaulty party hears key suggestions from all nonfaulty parties, it accepts them and picks some key. Secondly, if some nonfaulty primary picks a key to propose, the *suggest* messages it receives guarantee that any nonfaulty party will receive enough supporting *proof* messages. This means that all nonfaulty parties eventually accept the primary's proposal, even if they are locked on some other value. In the following lemmas assume that the number of faulty parties is $f < \frac{n}{3}$.

2.1 Safety Lemmas

The following lemma and corollary show that a primary cannot equivocate in a given view. More precisely, in a given view all nonfaulty parties send messages that report the same value, other than *echo* messages which might have more than one value. The proofs of the lemma and corollary consist of simple counting arguments and are omitted.

Algorithm 3 accept_key(key,value,proofs).

```
 1: $supporting \leftarrow 0$
 2: for all $(k, v, pk) \in proofs$ do
 3:     if $key \leq pk$ then
 4:         $supporting \leftarrow supporting + 1$
 5:     else if $key \leq k \land value = v$ then
 6:         $supporting \leftarrow supporting + 1$
 7: if $supporting \geq f + 1$ then
 8:     return $true$
 9: else
10:     return $false$
```

Algorithm 4 send_all_upon_join(message).

```
Code for party $i$:
 1: for all parties $j \in [n]$ do
 2:     upon $highest\_request[j] = view$, do
 3:         send $message$ to party $j$
```

▶ **Lemma 3.** *If two nonfaulty parties send the messages $\langle key1, val, v\rangle$ and $\langle key1, val', v\rangle$, then $val = val'$.*

▶ **Corollary 4.** *If two nonfaulty parties i and j send a $\langle tag, val, v\rangle$ and $\langle tag', val', v\rangle$ message such that $tag, tag' \in \{key1, key2, key3, lock\}$ then $val = val'$.*

The following lemma and corollary now show that all *done* messages that nonfaulty parties send have the same value. There are two ways nonfaulty party might send a *done* message: in the end of a view, or after receiving enough *done* messages from other parties. In the first view a nonfaulty party sends a *done* message in line 29, no nonfaulty party sends a *done* message with another value because of the previous non-equivocation claims. Then, once such a *done* message is sent, there are $f + 1$ nonfaulty parties that are locked on that value, and won't allow any other value to be proposed by a primary. Since all nonfaulty parties send *done* messages with the same value at the end of views, they never receive enough *done* messages with another value for them to echo that *done* message.

▶ **Lemma 5.** *If two nonfaulty parties send the messages $\langle done, val\rangle$ and $\langle done, val'\rangle$ in line 29, then $val = val'$.*

▶ **Corollary 6.** *If two nonfaulty parties send the messages $\langle done, val\rangle$ and $\langle done, val'\rangle$, then $val = val'$.*

The proofs of Lemma 5 and Corollary 6 closely follow the above description and are omitted.

2.2 Liveness Lemmas

The first two lemmas show that no nonfaulty party gets “stuck” in a view. If some nonfaulty party terminates, then every nonfaulty party eventually terminates as well. In addition, after GST, all nonfaulty parties start participating in consecutive views until terminating.

Algorithm 5 check_progress().

```
Code for party $i$:
 1: $\forall j \in [n]$  $highest\_abort[j] \leftarrow 0$
 2: upon receiving a $\langle request, v\rangle$ message from party $j$, do
 3:     if $highest\_request[j] < v$ then
 4:         $highest\_request[j] \leftarrow v$
 5: upon receiving a $\langle done, val\rangle$ message from $f+1$ parties with the same $val$, do
 6:     if no $done$ message has been previously sent then
 7:         send $\langle done, val\rangle$ to every party $j \in [n]$
 8: upon receiving a $\langle done, val\rangle$ message from $n-f$ parties with the same $val$, do
 9:     decide val and terminate
10: upon receiving an $\langle abort, v\rangle$ message from party $j$, do
11:     if $highest\_abort[j] < v$ then
12:         $highest\_abort[j] \leftarrow v$
13:         let $u$ be the $f+1$'th largest value in $highest\_abort$
14:         if $u > highest\_abort[i]$ then
15:             send $\langle abort, u\rangle$ to every party $j \in [n]$
16:             $highest\_abort[i] \leftarrow u$
17:         let $w$ be the $n-f$'th largest value in $highest\_abort$
18:         if $w \geq view$ then
19:             $view \leftarrow w+1$
```

▶ **Lemma 7.** *Observe some nonfaulty party i that terminates. All nonfaulty parties terminate no later than 2Δ time after both GST occurs, and i terminates.*

▶ **Lemma 8.** *Let v be the highest view that some nonfaulty party is in at GST. For every view $v' > v$, all nonfaulty parties either start view v', or terminate in some earlier view.*

Furthermore, if some nonfaulty party starts view v' after GST, all nonfaulty parties either terminate or start view v' no later than 2Δ time afterwards.

The proofs are straightforward and are omitted. Eventually, all nonfaulty parties participate in some view with a nonfaulty primary, if they haven't terminated previously. The next lemmas show that once that happens, all nonfaulty parties terminate. First of all, in order for that to happen, a primary needs to receive enough suggestions for a $key3$ that it will accept. The following lemma shows that every nonfaulty party's $key3$ field has enough support from nonfaulty parties for the primary to accept the key. Intuitively, since a nonfaulty party set its $key3$ field to some value, there exist $f+1$ nonfaulty parties that sent a $key2$ message with that value. The lemma shows that those $f+1$ nonfaulty parties have $key2$, $key2\_val$ and $prev\_key2$ fields that continue to support the key.

▶ **Lemma 9.** *If some nonfaulty party sets $key3 = v$, $key3\_val = val$ in view v, then there exist $f+1$ nonfaulty parties whose suggest messages in every view $v' > v$ support $key3$ and $key3\_val$.*

Proof. We will prove by induction that there exist $f+1$ nonfaulty parties for whom in every $v' > v$ either $prev\_key2 \geq key3$, or $key2 \geq key$ and $key2\_value = val$. Since those are the fields that nonfaulty parties send in *suggest* messages, that proves the lemma. First, observe view v. In that view, some nonfaulty party set $key3 = v$ and $key3\_val = val$. This means

Algorithm 6 process_messages(view).

```
Code for party $i$:
1:  $proofs \leftarrow \emptyset$
2:  upon receiving the first $\langle proof, k1, v1, pk1, view\rangle$ message from $j$, do
3:      if $view > k1 > pk1$ then
4:          add $(k1, v1, pk1)$ to $proofs$
5:  upon receiving the first $\langle propose, key, val, view\rangle$ message from $primary$, do
6:      if $lock = 0 \vee val = lock\_val$ then
7:          $send\_all\_upon\_join(\langle echo, val, view\rangle)$
8:      else if $view > key \geq lock$ then
9:          upon $open\_lock\,(proofs) = true$, do
10:             $send\_all\_upon\_join(\langle echo, val, view\rangle)$
11: upon receiving an $\langle echo, val, view\rangle$ message from $n - f$ parties with the same $val$, do
12:     $send\_all\_upon\_join(\langle key1, val, view\rangle)$
13:     if $key1\_val \neq val$ then
14:         $prev\_key1 \leftarrow key1, key1\_val \leftarrow val$
15:     $key1 \leftarrow view$
16: upon receiving a $\langle key1, val, view\rangle$ message from $n - f$ parties with the same $val$, do
17:     $send\_all\_upon\_join(\langle key2, val, view\rangle)$
18:     if $key2\_val \neq val$ then
19:         $prev\_key2 \leftarrow key2, key2\_val \leftarrow val$
20:     $key2 \leftarrow view$
21: upon receiving a $\langle key2, val, view\rangle$ message from $n - f$ parties with the same $val$, do
22:     $send\_all\_upon\_join(\langle key3, val, view\rangle)$
23:     $key3 \leftarrow view, key3\_val \leftarrow val$
24: upon receiving a $\langle key3, val, view\rangle$ message from $n - f$ parties with the same $val$, do
25:     $send\_all\_upon\_join(\langle lock, val, view\rangle)$ to every party $j \in [n]$
26:     $lock \leftarrow view, lock\_val \leftarrow val$
27: upon receiving a $\langle lock, val, view\rangle$ message from $n - f$ parties with the same $val$, do
28:     if no $done$ message has been previously sent then
29:         send $\langle done, val\rangle$ to every party $j \in [n]$
```

that it received a $\langle key2, val, v\rangle$ message from $n - f$ parties, $f + 1$ of whom are nonfaulty. In addition to other possible updates, every one of those parties updates $key2 = view$, and $key2\_val = val$ if that isn't true already. Those $f + 1$ parties prove the claim for view v.

Now assume the claim holds for every $v'' < v'$. Observe party j, which is one of the $f + 1$ parties described in the induction claim. If j doesn't update any of its $key2$ fields in view v', those conditions continue to hold in the end of view v' and in the beginning of the next view. If j only updates $key2$ to be v', then if $prev\_key2 \geq key3$, it remains that way, and if $key2 \geq key3$ as well as $key2\_val = key3\_val$, after updating $key2$ to be $v' > key2 \geq key3$, it also remains that way. Otherwise j updates $prev\_key2 = key2$ too. Note that $key2 > prev\_key2$ at all times. Therefore, before updating $prev\_key2$, regardless of which part of the induction claim holds, $key2 \geq key3$. After updating $prev\_key2$ to be $key2$, $prev\_key2 \geq key3$, completing the proof. ◀

The following lemma is used to show that if a nonfaulty primary chose some key, and some nonfaulty party has a lock, it is either the case that the key's value equals the lock's value, or there are enough nonfaulty parties that support opening the lock. Note that the conditions of

Algorithm 7 open_lock(proofs).

```
Code for party $i$:
1: $supporting \leftarrow 0$
2: for all $(k, v, pk) \in proofs$ do
3:     if $lock \leq pk$ then
4:         $supporting \leftarrow supporting + 1$
5:     else if $lock \leq k \wedge v \neq lock\_val$ then
6:         $supporting \leftarrow supporting + 1$
7: if $supporting \geq f + 1$ then
8:     return $true$
9: else
10:    return $false$
```

the lemma are nearly identical to the conditions the primary checks before accepting a proof as supporting some key. This means that before accepting a key, the primary essentially checks if there is enough support to open any other lock. Similarly to the previous lemma, this lemma shows that if some nonfaulty party sets $key2$ to some value, there are $f + 1$ parties that sent a $key1$ message with that value. Those $f + 1$ parties' $key1$, $key1\_val$ and $prev\_key1$ fields then continue to support any lock set previously with another value.

▶ **Lemma 10.** *Let $lock > 0$ be some nonfaulty party's lock and $lock\_val$ be its value. If some nonfaulty party either has $prev\_key2 \geq lock$ or $key2 \geq lock$ and $key2\_val \neq lock\_val$, then there exist $f + 1$ nonfaulty parties whose $key1$, $key1\_val$ and $prev\_key1$ fields support opening the lock.*

Proof. Let i be a nonfaulty party such that either $prev\_key2 \geq lock$ or $key2 \geq lock$ and $key2\_val \neq lock\_val$. If $key2 \geq lock > 0$ and $key2\_val \neq lock\_val$, i received a $\langle key1, key1\_val, key2\rangle$ message from $n - f$ parties in view $key2$. Out of those $n - f$ parties, at least $f + 1$ are nonfaulty. On the other hand, if $prev\_key2 \geq lock > 0$, then for some pair of values val, val' such that $val \neq val'$, i received a $\langle key1, val, prev\_key2\rangle$ message from $f + 1$ nonfaulty parties in view $prev\_key2$ and a $\langle key1, val', key2\rangle$ message from $f + 1$ nonfaulty parties in view $key2 > prev\_key2 \geq lock$. At least one of the values val, val' must not equal $lock\_val$ because $val \neq val'$. In other words, in both cases there exist $f + 1$ nonfaulty parties that sent a $\langle key1, val, v\rangle$ in view v such that $val \neq lock\_val$ and $v \geq lock$. Let I be the set of those nonfaulty parties.

We now prove by induction that for every $v' \geq v$, all of the parties in I either have $prev\_key1 \geq lock$ or $key1 \geq lock$ and $key1\_val \neq lock\_val$. First, observe view v. As stated above, in view v all of the parties in I sent a $\langle key1, val, v\rangle$ and thus set $key1 = v \geq lock$ and $key1\_val = val \neq lock\_val$, if it wasn't already so. Now, assume the claim holds for all views $v'' < v'$. Note that the values of $key1$ and $prev\_key1$ only grow throughout the run. This means that if $prev\_key1 \geq lock$ in the beginning of view v', this will also be true at the end of view v'. On the other hand, if that is not the case, then in the beginning of view v', $key1 \geq lock$ and $key1\_val \neq lock\_val$. If the value of $key1\_val$ isn't updated in view v', then $key1$ can only grow and thus the claim continues to hold. On the other, if the value of $key1\_val$ is updated in view v', then for some $val' \neq val$ the following updates take place: $key1 \leftarrow v', key1\_val \leftarrow val', prev\_key1 \leftarrow key1$. By assumption, in the beginning of view v', $key1 \geq lock$, and thus after the update $prev\_key1 \geq lock$, completing the proof. ◀

This final lemma ties the two previous lemmas together. Once a nonfaulty party is chosen as primary after GST, the primary receives enough keys, and each one of them has enough support to be accepted. Then, after the key is sent, every nonfaulty party either has a lock with the same value, or there is enough support to open its lock. From this point on, the view progresses easily and all nonfaulty parties terminate.

▶ **Lemma 11.** *Let v be the first view with a nonfaulty primary that starts after GST*[1]*. All nonfaulty parties decide on a value and terminate in view v, if they haven't done so earlier.*

Furthermore, if all messages between nonfaulty parties are actually delayed only δ time until being received, they decide on a value and terminate in $O(\delta)$ time.

The proof of the lemma follows naturally from the previous lemmas and is omitted.

2.3 Main Theorem

Using the previous lemmas, it is now possible to prove the main theorem:

▶ **Theorem 12.** *Algorithm 1 is a Byzantine Agreement protocol in partial synchrony resilient to $f < \frac{n}{3}$ Byzantine parties.*

Proof. We prove each property individually.

Correctness. Observe two nonfaulty parties i, j that decide on the values val, val' respectively. Party i first received a $\langle done, val\rangle$ message from $n-f$ parties, and j received a $\langle done, val'\rangle$ message from $n-f$ parties. Since $n-f > f$, i and j receive at least one of their respective messages from some nonfaulty party. From Corollary 6, all nonfaulty parties that send a *done* message do so with the same value. Therefore, $val = val'$.

Validity. Assume that all parties are nonfaulty and that they have the same input val. We will prove by induction that for every view v, every nonfaulty party has $key3\_val = val$. Furthermore, if some nonfaulty party sends a $\langle key1, val', v\rangle$ message, then $val' = val$. First, all parties set $key3\_val$ to be val in the beginning of the protocol. Assume the claim holds for every $v' < v$. In the beginning of view v, the primary calls the *view_change* protocol. Before completing *view_change*, the primary receives *suggest* messages from $n - f$ parties with their $key3\_val$ field. Since all parties are nonfaulty, they all send the $key3\_val$ they have at that point, and from the induction hypothesis $key3\_val = val$. This means that if the primary completes the *view_change* protocol, it sees that for every $(key, key\_val) \in suggestions$, $key\_val = val$ and thus if the primary sends a *propose* message it sends the message $\langle propose, val, key, v\rangle$ to all parties. Now, every nonfaulty party that sends a $key1$ message sends the message $\langle key1, val, v\rangle$. From Corollary 4, every nonfaulty party that sends a $\langle key3, val', v\rangle$ message, does so with $val' = val$. If a nonfaulty party updates $key3\_val$ to a new value val', it also sends a $\langle key3, val', v\rangle$ message. However, as shown above the only value sent in such a message is val so no nonfaulty party updates its $key3\_val$ field to any other value. Using Corollary 4, every nonfaulty party that sends a *lock* message does so with the value val. This means that any party that sends a *done* message in line 29, does so with the value val. Clearly any party that sends a *done* message in line 7 does so with the value val as well, because it never receives *done* messages with any other value. Finally, this means that every nonfaulty party that decides on a value decides on val.

[1] More precisely, by "starting after GST", we mean that the first time some nonfaulty party has $view \geq v$ is after GST.

Termination. Observe the system after GST, and let v be the highest view that some nonfaulty party is in at that time. From Lemma 8, all nonfaulty parties either terminate or participate in every view $v' > v$. Since the primaries are chosen in a round-robin fashion, after no more than $f + 1$ views, some nonfaulty party starts a view with a nonfaulty primary. From Lemma 11, all nonfaulty parties either terminate in that view or earlier. ◀

2.4 Complexity Measures

The main complexity measures of interest are round complexity, word complexity, and space complexity.

Word complexity. In IT-HS in every round, every party sends at most $O(1)$ words to every other party. We assume that a *word* is large enough to contain any counter or identifier. This implies that just $O(n^2)$ words are sent in each round.

Round complexity. As IT-HS is a primary-backup view-based protocol (like Paxos and PBFT), there are no bounds on the number of rounds while the system is still asynchronous. Therefore, we use the standard measure of counting the number of rounds and number of words sent after GST. Furthermore, in order to be useful in the task of agreeing on many values, a desirable property is *optimistic responsiveness*: when the primary is nonfaulty and the network delay is low, all nonfaulty parties complete the protocol at network speed. This desire is captured in the next definition:

▶ **Definition 13** (Optimistic Responsivness). *Assume all messages between nonfaulty parties are actually delivered in $\delta < \Delta$ time. The protocol is said to be* optimistically responsive *if all nonfaulty parties complete the protocol in $O(\delta)$ time after a nonfaulty primary is chosen after GST.*

Space complexity. We separate the local space complexity into two types: persistent memory and transient memory. In this setting, parties can crash and be rebooted. Persistent memory is never erased, even in the event of a crash, while transient memory can be erased by a reboot event. IT-HS requires asymptotically optimal $O(1)$ persistent storage (measured in words) and just $O(1)$ transient memory per communication channel (so a total of $O(n)$ transient memory).

After a reboot, nonfaulty parties can ask other parties to send messages that help recover information needed in their transient memory. In this setting we assume that all nonfaulty parties that terminate still reply to messages asking for previously sent information.

▶ **Theorem 14.** *During Algorithm 1, each nonfaulty party sends a constant number of words to each other party in each view and requires $O(n)$ memory overall, out of which $O(1)$ is persistent memory. Furthermore, the protocol is optimistically responsive.*

Proof. First note that each view consists of one message sent from all parties to the primary, one message sent from the primary to all parties, and a constant number of all-to-all communication rounds. In addition, each message consists of no more than 7 words. Overall, each party only sends a constant number of messages to every party, each with a constant number of words. In each view, every nonfaulty party needs to remember which messages were sent to it by other parties, as well as a constant amount of information about every *suggest* and *proof* message. Since a constant number of words and messages is sent from each party to every other party, this requires $O(n)$ memory. Note that once a new view is started, all of the information stored in the previous call to *view_change* and *process_messages* is freed. Other than that, every nonfaulty party allocates two arrays of size n, a constant number of other fields, and needs to remember the first *done* messages received from every other party. This also requires $O(n)$ memory. Overall, the only fields that need to be stored in persistent memory are the *view*, *lock*, *lock_val*, and various *key*, *key_val* and *prev_key* fields, as well as the messages it sent in the current view, and the last *done*, *request* and *abort* messages it sent. This is a constant number of fields, in addition to a constant number of messages. After being rebooted, a nonfaulty party i can ask to receive the last *done*, *request*, and *abort* messages sent by all nonfaulty parties to restore the information it lost that doesn't pertain to any specific view, and any message sent in the current view. In addition, it sends a *request* message for its current view. Upon receiving such a message, a nonfaulty party j replies with the last *done*, *request* and *abort* messages it sent. In addition, if j is in the view that party i asked about, it also re-sends the messages it sent in the current view. Note that this is essentially the same as i receiving messages late and starting its view after being rebooted, and thus all of the properties still hold. The fact that the protocol is optimistically responsive is proven in Lemma 11. ◀

3 Multi-Shot Byzantine Agreement and State Machine Replication

This section describes taking a Byzantine Agreement protocol and using it to solve two tasks that are natural extensions of a single shot agreement. Both tasks deal with different formulations for the idea of agreeing on many values, instead of just one.

3.1 State Machine Replication with Stable Leader (a la PBFT)

In the task of State Machine Replication [17], all parties (called replicas) have knowledge of the same state machine. Each party receives a (possibly infinite) series of instructions to perform on the state machine as input. The goal of the parties is to all perform the same actions on the state machine in the same order. More precisely, the parties are actually only interested in the state of the state machine, and aren't required to see all of the intermediary states throughout computation. In order to avoid trivial solutions, if all parties are nonfaulty and they have the same s'th instruction as input, then they all execute it as the s'th instruction for the state machine. This task can be achieved utilizing any Byzantine Agreement protocol, using ideas from the PBFT protocol.

In addition to the inputs, the protocol is parameterized by a window size α. All parties participate in α instances of the Byzantine Agreement protocol, each one tagged with the current decision number. After each decision, every party saves a log of their current decision, and updates the state machine according to the decided upon instruction. Then, after every $\frac{\alpha}{2}$ decisions, each party saves a "checkpoint" with the current state of the state machine, and deletes the log of the $\frac{\alpha}{2}$ oldest decisions. Then, before starting the next $\frac{\alpha}{2}$ decisions, every party sends its current checkpoint and makes sure it receives the same state from $n-f$ parties using techniques similar to Bracha broadcast. Furthermore, as long as no view fails, the primary isn't replaced. This means that eventually at some point, either there exists a faulty primary that always acts like a nonfaulty primary, or a nonfaulty primary is chosen and is never replaced. Both sending the checkpoints and replacing faulty leaders require more implementation details which can be found in [9].

Using these techniques, all parties can decide on $O(\alpha)$ instructions at a time, improving the throughput of the algorithm. The communication complexity per view remains similar to the communication complexity of the IT-HS algorithm, but once a nonfaulty primary is reached after GST, all invocations of the protocol require only one view to terminate. Alternatively, if a nonfaulty primary is never reached after GST, a faulty party acts like a nonfaulty primary indefinitely, which yields the same round complexity. Finally, if we assume that a description of the state machine requires $O(S)$ space, the protocol now requires $O(S+\alpha)$ persistent space in order to store the checkpoints and store the $O(1)$ state for each slot in the window. In addition, the protocol requires $O(\alpha \cdot n + S)$ transient space in order to store the information about all active calls to IT-HS, the α decisions in the log, and a description of the current state of the state machine.

3.2 Multi-Shot Agreement with Pipelining (a la HotStuff)

In contrast, we can take the approach of HotStuff [18] and solve the task of multi-shot agreement. In this task, party i has an infinite series of inputs $x_i^1, x_i^2, \ldots$, and the goal of the parties is to agree on an infinite number of values. Each decision is associated with a slot which is the number $s \in \mathbb{N}$ of the decision made. Each one of these decisions is required to have the agreement properties, i.e.: eventually all nonfaulty parties decide on a value for slot s, they all decide on the same value, and if all parties are nonfaulty and have the same input val for slot s, the decision for the slot is val.

A naive implementation for this task is to sequentially call separate instances of IT-HS for every slot $s \in \mathbb{N}$, each with the input (s, x_i^s). In order to improve the throughput of the protocol, after completing an instance of the IT-HS protocol, the parties can continue with the next view and the next primary in the round-robin. This slight adjustment ensures that after GST, $n-f$ out of every n views have a decision made, and if messages between nonfaulty parties are only delayed δ time, each one of those views requires only $O(\delta)$ time to reach a decision. Slight adjustments need to be made in that case so that *abort* messages are sent about views regardless of the slot, so that all parties continue participating in the same views throughout the protocol. In addition, messages about different slots need to be ignored.

In the case of the optimistic assumption that most parties are nonfaulty, a significantly more efficient alternative can be gleaned from the HotStuff protocol. This alternative uses a technique called *pipelining* (or chaining). Roughly speaking, in this technique, all parties start slot s by appending messages, starting on the second round (round, not view) of slot $s-1$. In the case of HT-IS, the protocol can be changed so that *suggest* messages are sent to all parties, and then each party starts slot s after receiving $n-f$ *suggest* messages in slot s. Note that the exact length of timeouts needs to be slightly adjusted, and the details can be found in [18]. In slot s, a nonfaulty primary appends its current proposal to the proposal it heard in slot $s-1$. Then, before deciding on a value in slot s, parties check that the decision values in the previous slots agree with the proposal in slot s. If they do, then the parties agree on the value in this slot as well. In this protocol, each view lasts for 11Δ time, so if at some point a primary sees that a proposal from 11 views ago failed, it appends its proposal to the first one that it accepted from a previous view. After GST, if there are $m+11$ nonfaulty primaries in a row, then the last m primaries are guaranteed to complete the protocol, and thus add m decisions in $(m+11)\Delta$ time instead of in $m \cdot 11\Delta$ time. This means that in the optimistic case that a vast majority of parties are nonfaulty, the throughput of this protocol is greatly improved as compared to the naive implementation. In this protocol the communication complexity per view is still $O(n^2)$ messages, but a larger number of words. However, note that it is not always the case that if a nonfaulty primary is chosen, its proposal is accepted. To obtain bounded memory requirement one needs to add a checkpointing mechanism, similar to PBFT. As in PBFT, only $O(n)$ transient space and $O(1)$ persistent space are required per decision in addition to the log of the decisions.

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 337–346, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331612.

2 Charles H. Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John Smolin. Experimental quantum cryptography. *Journal of Cryptology*, 5(1):3–28, January 1992. doi:10.1007/BF00191318.

3 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, November 1987. doi:10.1016/0890-5401(87)90054-X.

4 Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. http://knowen-production.s3.amazonaws.com/uploads/attachment/file/1814/Buchman_Ethan_201606_Msater%2Bthesis.pdf, 2016.

5 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus, 2018. arXiv:1807.04938.

6 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget, 2017. arXiv:1710.09437.

7 Christian Cachin. Yet another visit to paxos. https://cachin.com/cc/papers/pax.pdf, 2010.

8 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, page 524–541, Berlin, Heidelberg, 2001. Springer-Verlag.

9 Miguel Castro. Practical byzantine fault tolerance. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf, 2001.

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

11 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

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

13 Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes, 2015. arXiv:1506.06817.

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

15 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, October 2007. doi:10.1145/1323293.1294267.

16 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

17 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.

18 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.