Cogsworth: Byzantine View Synchronization

Oded Naor, Technion and Calibra
Mathieu Baudet, Calibra
Dahlia Malkhi, Calibra
Alexander Spiegelman, VMware Research

Most methods for Byzantine fault tolerance (BFT) in the partial synchrony setting divide the local state of the nodes into views, and the transition from one view to the next dictates a leader change. In order to provide liveness, all honest nodes need to stay in the same view for a sufficiently long time. This requires *view synchronization*, a requisite of BFT that we extract and formally define here.

Existing approaches for Byzantine view synchronization incur quadratic communication (in n, the number of parties). A cascade of $O(n)$ view changes may thus result in $O(n^3)$ communication complexity. This paper presents a new Byzantine view synchronization algorithm named Cogsworth, that has optimistically linear communication complexity and constant latency. Faced with benign failures, Cogsworth has expected linear communication and constant latency.

The result here serves as an important step towards reaching solutions that have overall quadratic communication, the known lower bound on Byzantine fault tolerant consensus. Cogsworth is particularly useful for a family of BFT protocols that already exhibit linear communication under various circumstances, but suffer quadratic overhead due to view synchronization.

1. INTRODUCTION

Logical synchronization is a requisite for progress to be made in asynchronous state machine replication (SMR). Previous Byzantine fault tolerant (BFT) synchronization mechanisms incur quadratic message complexities, frequently dominating over the linear cost of the consensus cores of BFT solutions. In this work, we define the *view synchronization* problem and provide the first solution in the Byzantine setting, whose latency is bounded and communication cost is linear, under a broad set of scenarios.

1.1. Background and Motivation

Many practical reliable distributed systems do not rely on network synchrony because networks go through outages and periods of Distributed Denial-of-Service (DDoS) attacks; and because synchronous protocols have hard coded steps that wait for a maximum delay. Instead, asynchronous replication solutions via state machine replication (SMR) [Schneider 1990] usually optimize for stability periods. This approach is modeled as partial synchrony [Dwork et al. 1988]. It allows for periods of asynchrony in which progress might be compromised, but consistency never does.

In the crash-failure model, this paradigm underlies most successful industrial solutions, for example, the Google Chubbie lock service [Burrows 2006], Yahoo's tcdZooKeeper [Hunt et al. 2010], etcd [etcd community 2019], Google's Spanner [Corbett et al. 2013], Apache Cassandra [Cassandra 2014] and others. The algorithmic cores of these systems, e.g., Paxos [Lamport et al. 2001], Viewstamp Replication [Oki and Liskov 1988], or Raft [Ongaro and Ousterhout 2014], revolve around a view-based paradigm. In the Byzantine model, this paradigm underlies many blockchain systems, including VMware's Concord [VMware 2019], Hyperledger Fabric [Androulaki et al. 2018], Cypherium [Guo 2017; Cypherium 2019], Celo [Kamvar et al. 2019], PaLa [Chan et al. 2018] and Libra [Libra Association Members 2019]. The algorithmic cores of these BFT system are view-based, e.g., PBFT [Castro et al. 1999], SBFT [Gueta et al. 2019], and HotStuff [Yin et al. 2019].

The advantage of the view-based paradigm is that each view has a designated leader that can drive a decision efficiently. Indeed, in both models, there are protocols that have per-view linear message and communication complexity, which is optimal.

In order to guarantee progress, nodes must give up when a view does not reach a decision after a certain timeout period. Mechanisms for changing the view whose communication is linear exist both for the crash model (all the above) and, recently, for the Byzantine model (HotStuff [Yin et al. 2019]). An additional requirement for progress is that all nodes overlap for a sufficiently long period. Unfortunately, all of the above protocols incur quadratic message complexity for view synchronization.

In order to address this, we first define the *view synchronization* problem independently of any specific protocol and in a fault-model agnostic manner. We then introduce a view synchronization algorithm called Cogsworth whose message complexity is linear in expectation, as well as in the worst-case under a broad set of conditions.

1.2. The View Synchronization Problem

We introduce the problem of *view synchronization*. All nodes start at view zero. A view change occurs as an interplay between the synchronizer, which implements a view synchronization algorithm and the outer consensus solutions. The consensus solution must signal that it wishes to end the current view via a *wish_to_advance*() notification. The synchronizer eventually invokes a consensus *propose_view*(v) signal to indicate when a new view v starts. View synchronization requires to eventually bring all honest nodes to execute the same view for a sufficiently long time, for the outer consensus protocol to be able to drive progress.

The two measures of interest to us are latency and communication complexity between these two events. Latency is measured only during periods of synchrony, when a bound δ on message transmission delays is known to all nodes, and is expressed in δ units.

View synchronization extends the PaceMaker abstraction presented in [Yin et al. 2019], formally defines the problem it solves, and captures it as a separate component. It is also related to the seminal work of Chandra and Toueg [Chandra et al. 1996], [Chandra and Toueg 1996] on *failure detectors*. Like failure detectors, it is an abstraction capturing the conditions under which progress is guaranteed, without involving explicit engineering solutions details such as packet transmission delays, timers, and computation. Specifically, Chandra and Toueg define a leader election abstraction, denoted Ω, where eventually all non-faulty nodes trust the same non-faulty node as the leader. Ω was shown to be the weakest failure detector needed in order to solve consensus. Whereas Chandra and Toueg's seminal work focuses on the possibility/impossibility of an eventually elected leader, here we care about how quickly it takes for a good leader to emerge (i.e., the latency), at what communication cost, and how to do so repeatedly, allowing the extension of one time single-shot consensus to SMR.

We tackle the view synchronization problem against asynchrony and the most severe type of faults, Byzantine [Lamport et al. 1982; Lamport 1983]. This makes the synchronizers we develop particularly suited for Byzantine Fault Tolerance (BFT) consensus systems relevant in today's crypto-economic systems.

More specifically, we assume a system of n nodes that need to form a sequence of *consensus* decisions that implement SMR. We assume up to $f < n/3$ nodes are Byzantine, the upper bound on the number of Byzantine nodes in which Byzantine agreement is solvable [Fischer et al. 1986]. The challenge is that during "happy" periods, progress might be made among a group of Byzantine nodes cooperating with a "fast" sub-group of the honest nodes. Indeed, many solutions advance when a leader succeeds in proposing a value to a quorum of $2f+1$ nodes, but it is possible that only the $f+1$ "fast" honest nodes learn it and progress to the next view. The remaining f "slow" honest nodes might stay behind, and may not even advance views at all. Then at some point, the f Byzantine nodes may stop cooperating. A mechanism is needed to bring the "slow" nodes to the same view as the $f+1$ "fast" ones.

Thus, our formalism and algorithms may be valuable for the consensus protocols mentioned above, as well as others, such as Casper [Buterin and Griffith 2017] and Tendermint [Buchman 2017; Buchman et al. 2018], which reported problems around liveness [Milosevic 2018; Pyrofex Corporation 2018].

1.3. View Synchronization Algorithms

We first extract two synchronization mechanisms that borrow from previous BFT consensus protocols, casting them into our formalism and analyzing them.

Table I. Comparison of the different protocols for view synchronization

	View doubling	**broadcast-based**		**Cogsworth: leader-based**		
				Fault type		
Communication complexity	0	expected	$O(n^2)$	Byzantine	optimal	$O(n)$
		worst-case	$O(t\cdot n^2)$		expected	$O(n^2)$
					worst-case	$O(t\cdot n^2)$
				Benign	expected	$O(n)$
					worst-case	$O(t\cdot n)$
Latency	unbounded	expected	$O(\delta)$	Byzantine + Benign	expected	$O(\delta)$
		worst-cast	$O(t\cdot\delta)$		worst-case	$O(t\cdot\delta)$

Note: t is the number of actual failures, and δ is the bound on message delivery after GST.

One is a straw-man mechanism that requires no communication at all and achieves synchronization albeit with unbounded latency. This synchronizer works simply by doubling the duration of each view. Eventually, it guarantees a sufficiently long period in which all the nodes are in the same view.

The second is the broadcast-based synchronization mechanism built into PBFT [Castro et al. 1999] and similar Byzantine protocols, such as [Gueta et al. 2019]. This synchronizer borrows from the Bracha reliable broadcast algorithm [Bracha 1987]. Once a node hears of $f+1$ nodes who wish to enter the same view, it relays the wish reliably so that all the honest nodes enter the view within a bounded time.

The properties of these synchronizers in terms of latency and communication costs are summarized in Table I. For brevity, these algorithms and their analysis are deferred to Appendix A.

Cogsworth: leader-based synchronizer

The main contribution of our work is Cogsworth[1], which is a leader-based view synchronization algorithm. Cogsworth utilizes views that have an honest leader to relay messages, instead of broadcasting them. When a node wishes to advance a view, it sends the message to the leader of the view, and not to all the other nodes. If the leader is honest, it will gather the messages from the nodes and multicast them using a threshold signature [Boneh et al. 2001; Cachin et al. 2005; Shoup 2000] to the rest of the nodes, incurring only a linear communication cost. The protocol implements additional mechanisms to advance views despite faulty leaders.

The latency and communication complexity of this algorithm depend on the number of actual failures and their type. In the best case, the latency is constant and communication is linear. Faced with t benign failures, in expectation, the communication is linear and in worst case $O(t\cdot n)$, as mandated by the lower bound of Dolev and Reischuk [Dolev and Reischuk 1985]; the latency is expected constant and $O(t\cdot\delta)$ in the worst-case. Byzantine failures do not change the latency, but they can drive the communication to an expected $O(n^2)$ complexity and in the worst-case up to $O(t\cdot n^2)$. It remains open whether a worst-case linear synchronizer whose latency is constant is possible.

To summarize, Cogsworth performs just as well as a broadcast-based synchronizer in terms of latency and message complexity, and in certain scenarios shows up-to $O(n)$ better results in terms of message complexity. Table I summarizes the properties of all three synchronizers.

1.4. Contributions

The contributions of this paper as follows:

- To the best of our knowledge, this is the first paper to formally define the problem of view synchronization.
- It includes two natural synchronizers algorithms cast into this framework and uses them as a basis for comparison.

[1]Cogsworth is the enchanted clock from Disney's classic "Beauty and the Beast".

— It introduces Cogsworth, a leader-based Byzantine synchronizer exhibiting faultless and expected linear communication complexity and constant latency.

Structure. The rest of this paper is structured as follows: §2 discusses the model; §3 formally presents the view synchronization problem; §4 presents the Cogsworth view synchronization algorithm with formal correctness proof latency and communication cost analysis; §5 describes real-world implementations where the view synchronization algorithms can be integrated; §6 presents related work; and §7 concludes the paper. The description of the two natural view synchronization algorithms, view doubling and broadcast-based, are presented in Appendix A.

2. MODEL

We follow the eventual synchronous model [Dwork et al. 1988] in which the execution is divided into two durations; first, an unbounded period of asynchrony, where messages do not have a bounded time until delivered; and then, a period of synchrony, where messages are delivered within a bounded time, denoted as δ. The switch between the first and second periods occurs at a moment named *Global Stabilization Time (GST)*. We assume all messages sent before GST arrive at or before $\mathrm{GST}+\delta$.

Our model consists of a set $\Pi = \{\mathcal{P}_i\}_{i=1}^{n}$ of n nodes, and a known mapping, denoted by $\mathrm{Leader}(\cdot)\colon\ \mathbb{N} \mapsto \Pi$ that continuously rotates among the nodes. Formally, $\forall j \geq 0\colon\ \bigcup_{i=j}^{\infty} \mathrm{Leader}(i) = \Pi$. We use a cryptographic signing scheme, a public key infrastructure (PKI) to validate signatures, as well as a threshold signing scheme [Boneh et al. 2001; Cachin et al. 2005; Shoup 2000]. The threshold signing scheme is used in order to create a compact signature of k-of-n nodes and is used in other consensus protocols such as [Cachin et al. 2005]. Usually $k = f+1$ or $k = 2f+1$.

We assume a non-adaptive adversary who can corrupt up to $f < n/3$ nodes at the beginning of the execution. This corruption is done without the knowledge of the mapping $\mathrm{Leader}(\cdot)$. The set of remaining $n-f$ honest nodes is denoted H. We assume the honest nodes may start their local execution at different times.

In addition, as in [Abraham et al. 2019; Cachin et al. 2005], we assume the adversary is polynomial-time bounded, i.e., the probability it will break the cryptographic assumptions in this paper (e.g., the cryptographic signatures, threshold signatures, etc.) is negligible.

3. PROBLEM DEFINITION

We define a *synchronizer*, which solves the view synchronization problem, to be a long-lived task with an API that includes a *wish_to_advance*() operation and a *propose_view*(v) signal, where $v \in \mathbb{N}$. Nodes may repeatedly invoke *wish_to_advance*(), and in return get a possibly infinite sequence of *propose_view*($\cdot$) signals. Informally, the synchronizer should be used by a high-level abstraction (e.g., BFT state-machine replication protocol) to synchronize view numbers in the following way: All nodes start in view 0, and whenever they wish to move to the next view they invoke wish_to_advance(). However, they move to view v only when they get a *propose_view*(v) signal.

Formally, a *time interval* $\mathcal{I}$ consists of a starting time t_1 and an ending time $t_2 \geq t_1$ and all the time points between them. $\mathcal{I}$'s length is $|\mathcal{I}| = t_2 - t_1$. We say $\mathcal{I}' \subseteq \mathcal{I}''$ if $\mathcal{I}'$ begins after or when $\mathcal{I}''$ begins, and ends before or when $\mathcal{I}''$ ends. We denote by $t_{\mathcal{P},v}^{prop}$ the time when node $\mathcal{P}$ gets the signal propose_view(v), and assume that all nodes get propose_view(0) at the beginning of their execution. We denote time $t = 0$ as the time when the last honest node began its execution, formally $\max_{\mathcal{P}\in H} t_{\mathcal{P},0}^{prop} = 0$. We further denote $\Delta_{\mathcal{P},v}^{exec}$ as the time interval in which node $\mathcal{P}$ is in view v, i.e., $\Delta_{\mathcal{P},v}^{exec}$ begins at $t_{\mathcal{P},v}^{prop}$ and ends at $t_{\mathcal{P},v}^{end} \triangleq \min_{v'>v}\left\{t_{\mathcal{P},v'}^{prop}\right\}$. We say node $\mathcal{P}$ is at view v at time t, or *executes view v at time t*, if $t \in \Delta_{\mathcal{P},v}^{exec}$.

We are now ready to define the two properties that any synchronizer must achieve. The first property, named *view synchronization* ensures that there is an infinite number of views with an honest leader that all the correct nodes execute for a sufficiently long time:

PROPERTY 1 (VIEW SYNCHRONIZATION). *For every $c \geq 0$ there exists $\alpha > 0$ and an infinite number of time intervals and views $\{\mathcal{I}_k, v_k\}_{k=1}^{\infty}$, such that if the interval between every two consecutive calls to* $\mathsf{wish\_to\_advance}()$ *by an honest node is α, then for any $k \geq 1$ and any $\mathcal{P} \in H$ the following holds:*

(1) $|\mathcal{I}_k| \geq c$
(2) $\mathcal{I}_k \subseteq \Delta^{exec}_{\mathcal{P},v_k}$
(3) $Leader(v_k) \in H$

The second property ensures that a synchronizer will only signal a new view if an honest node wished to advance to it. Formally:

PROPERTY 2 (SYNCHRONIZATION VALIDITY). *The synchronizer signals* $\mathsf{propose\_view}(v')$ *only if there exists an honest node $\mathcal{P} \in H$ and some view v s.t. $\mathcal{P}$ calls* $\mathsf{wish\_to\_advance}()$ *at least $v' - v$ times while executing view v.*

Discussion. The parameter α, which is used in Property 1 is the time an honest node waits between two successive invocations of $\mathsf{wish\_to\_advance}()$, and may differ between view synchronization algorithms. This parameter is needed to make sure that $\mathsf{wish\_to\_advance}()$ is called an infinite number of times in an infinite run. In reality, it is likely that in most view synchronization algorithms α is larger than some value d which is a function of the message delivery bound δ, and also of c from Property 1, i.e., the synchronization algorithm will work for any $\alpha \geq d(\delta, c)$. In this case, a consensus protocol using the synchronizer can execute the same view as long as progress is made, and trigger a new view synchronization in case liveness is lost. See Appendix A.3 for concrete examples.

The requirement that the leader of all the synchronized views is honest is needed to ensure that once a view is synchronized, the leader of that view will drive progress in the upper layer protocol, thus ensuring liveness. Without this condition, a synchronizer might only synchronize views with faulty leaders.

Synchronization validity (Property 2) ensures that the synchronizer does not suggest a new view to the upper-layer protocol unless an honest node running that upper-layer protocol wanted to advance to that view.

Latency and communication complexity. In order to define how the latency and message communication complexity are calculated, we first define $\mathcal{I}_k^{start}$ to be the time at which the *k-th view synchronization is reached*. Formally, $\mathcal{I}_k^{start} \triangleq \max_{\mathcal{P} \in H} \left\{ t^{prop}_{\mathcal{P},v_k} \right\}$, where v_k is defined according to Property 1.

With this we can define the latency of a synchronizer implementation:

Definition 3.1 (*Synchronizer latency*). The latency of a synchronizer is defined as $\lim_{n\to\infty} \left((\mathcal{I}_1^{start} - \mathrm{GST}) + \sum_{k=2}^{n} \mathcal{I}_k^{start} - \mathcal{I}_{k-1}^{start} \right) / n$.

Next, in order to define communication complexity, we first need to introduce a few more notations. Let $M_{\mathcal{P},v_1 \to v_2}$ be the total number of messages $\mathcal{P}$ sent between $t^{prop}_{\mathcal{P},v_1}$ and $t^{prop}_{\mathcal{P},v_2}$. In addition, denote $M_{\mathcal{P},\to v}$ as the total number of messages sent by $\mathcal{P}$ from the beginning of $\mathcal{P}$'s execution and $t^{prop}_{\mathcal{P},v}$.

With this, we define the communication complexity of a synchronizer implementation:

Definition 3.2 (*Synchronizer communication complexity*). Denote v_k the k-th view in which view synchronization occurs (Property 1). The message communication cost of a synchronizer is defined as $\lim_{n\to\infty} \left(\sum_{\mathcal{P}\in H} M_{\mathcal{P},\to v_1} + \sum_{k=2}^{n} \sum_{\mathcal{P}\in H} M_{\mathcal{P},v_{k-1}\to v_k} \right) / n$.

This concludes the formal definition of the view synchronization problem. Next, we present Cogsworth, a view synchronization algorithm with expected constant latency and linear communication complexity in a variety of scenarios.

Algorithm 1: Cogsworth: Leader-based synchronizer for node $\mathcal{P}$

```
1  initialize :
2      $curr \leftarrow 0$
3      $attemptedTC \leftarrow 0$
4      $attemptedQC \leftarrow 0$

   Every node:
5  on wish_to_advance():
6      send "WISH, $curr+1$" to Leader($curr+1$)
7  upon receiving a valid "TC, $v$" from Leader($r$) s.t. $v \leq r \leq v+f+1$:
8      send "TC, $v$" to Leader($v$)    /* Ensures that if Leader($v$) is honest then latency is linear */
9      send "VOTE, $v$" to Leader($r$)
10 after $2\delta$ from last sending "WISH, $v$" and not receiving "TC, $v$" and $attemptedTC \leq curr+f+1$:
11     send "WISH, $v$" to Leader($attemptedTC$)
12     $attemptedTC \leftarrow attemptedTC+1$
13 upon receiving a valid "QC, $v$" from Leader($r$) s.t. $v \leq r \leq v+f+1$:
14     $attemptedTC \leftarrow v$
15     $attemptedQC \leftarrow v$
16     $curr \leftarrow v$
17     propose_view($v$)
18 after $2\delta$ from last sending "VOTE, $v$" and not receiving "QC, $v$" and $attemptedQC \leq curr+f+1$:
       /* Also need to send "TC, $v$" so that the next leader can multicast to the rest */
19     send "VOTE, $v$" and "TC, $v$" to Leader($attemptedQC$)
20     $attemptedQC \leftarrow attemptedQC+1$

   Leader (Leader($r$) = $\mathcal{P}$):
21 upon receiving $f+1$ "WISH, $v$" or "TC, $v$" s.t. $r-(f+1) \leq v \leq r$ and not sending "TC, $v$" before:
22     multicast "TC, $v$" with a threshold signature to all nodes (including self)
23 upon receiving $2f+1$ "VOTE, $v$" messages s.t. $r-(f+1) \leq v \leq r$ and not sending "QC, $v$" before:
24     multicast "QC, $v$" with a threshold signature to all nodes (including self)
```

4. COGSWORTH: LEADER-BASED SYNCHRONIZER

Before presenting Cogsworth, it is worth mentioning that we assume that all messages between nodes are signed and verified; for brevity, we omit the details about the cryptographic signatures. In the algorithm, when a node collects messages from x senders, it is implied that these messages carry x distinct signatures. We also assume that the Leader$(\cdot)$ mapping is based on a permutation of the nodes such that every consecutive $f+1$ views have at least one honest leader, e.g., Leader$(v) = (v \bmod n)+1$. The algorithm can be easily altered to a scenario where this is not the case.

4.1. Overview

Cogsworth is a new approach to view synchronization that leverages leaders to optimistically achieve linear communication. The key idea is that instead of nodes broadcasting synchronization messages all-to-all and incurring quadratic communication, nodes send messages to the leader of the view they wish to enter. If the leader is honest, it will relay a single broadcast containing an aggregate of all the messages it received, thus incurring only linear communication.

If a leader of a view v is Byzantine, it might not help as a relay. In this case, the nodes time out and then try to enlist the leaders of subsequent views, one by one, up to view $v+f+1$, to help with relaying. Since at least one of those leaders is honest, one of them will successfully relay the aggregate.

The full protocol is presented in Alg. 1, and is consisted of several message types. The first two are sent from a node to a leader. They are used to signal to the leader that the node is ready to advance to the next stage in the protocol. Those messages are named " WISH, v" and " VOTE, v" where v is the view the message refers to.

The other two message types are ones that are sent from leaders to nodes. The first is called " TC, v" (short for "Time Certificate") and is sent when the leader receives $f+1$ " WISH, v" messages; and the second is called " QC, v" (short for "Quarum Certificate") and is sent when the leader receives $2f+1$ " VOTE, v" messages. In both cases, a leader aggregates the messages it receives using threshold signatures such that each broadcast message from the leader contains only one signature.

The general flow of the protocol is as follows: When $\mathsf{wish\_to\_advance}()$ is invoked, the node sends " WISH, v" to Leader(v), where v is the view succeeding *curr* (Line 5). Next, there are two options: (i) If Leader(v) forms a " TC, v" , it broadcast it to all nodes (Line 21). The nodes then respond with " VOTE, v" message to the leader (Line 7) (ii) Otherwise, if 2δ time elapses after sending " WISH, v" to Leader(v) without receiving " TC, v" , a node gives up and sends " WISH, v" to the next leader, i.e., Leader$(v+1)$ (Line 10). It then waits again 2δ before forwarding " WISH, v" to Leader$(v+2)$, and so on, until " TC, v" is received.

Whenever " TC, v" has been received, a node sends " VOTE, v" (even if it did not send " WISH, v") to Leader(v). Additionally, as above, it enlists leaders one by one until " QC, v" is obtained. Here, the node sends leaders " TC, v" as well as " VOTE, v". When a node finally receives " QC, v" from a leader, it enters view v immediately (Line 13).

Correctness. We will prove that Cogsworth achieves eventual view synchronization (Property 1) for any $\alpha \geq 4\delta$ as well as synchronization validity (Property 2). Thus, the claims and lemmas bellow assume this.

We start by proving that if an honest node entered a new view, and the leader of that view is honest, then all the other honest nodes will also enter that view within a bounded time.

CLAIM 1. *After GST, if an honest node enters view v at time t, and the leader of view v is honest then all the honest nodes enter view v by $t+4\delta$, i.e., if Leader$(v) \in H$ then* $\max_{\mathcal{P}_i \in H}\left\{t^{prop}_{\mathcal{P}_i,v}\right\} - \min_{\mathcal{P}_j \in H}\left\{t^{prop}_{\mathcal{P}_j,v}\right\} \leq 4\delta$.

PROOF. Let $\mathcal{P}_i$ be the first honest node that entered view v at time t. $\mathcal{P}_i$ entered view v since it received " QC, v" from Leader(r) such that $v \leq r \leq v+f+1$ (Line 13).

If $r = v$ then we are done, since when Leader(v) sent " QC, v" it also sent it to all the other honest nodes (Line 24), which will be received by $t+\delta$, and all the honest nodes will enter view v.

Next, if $r > k$ then the only way for Leader(v) to send " QC, v" is if it gathered $2f+1$ " VOTE, v" messages (Line 23), meaning at least $f+1$ of the " VOTE, v" messages were sent by honest nodes. An honest node will send a " VOTE, v" message only after first receiving " TC, v" from Leader(r') s.t. $v \leq r' \leq v+f+1$ (Line 7).

Since when receiving a " TC, v" an honest node sends the " TC, v" to Leader(v) (Line 8), then Leader(v) will receive " TC, v" by $t+\delta$, will forward it to all other nodes by $t+2\delta$, who will send " VOTE, v" to Leader(v) by $t+3\delta$ and by $t+4\delta$ all honest nodes will receive " QC, v" from Leader(v) and enter view v. □

Next, assuming an honest node entered a new view, we bound the time it takes to at least $f+1$ honest nodes to enter the same view. Note that this time we do not assume anything on the leader of the new view, and it might be Byzantine.

CLAIM 2. *After GST, when an honest node enters view v at time t, at least $f+1$ honest nodes enter view v by $t+2\delta(f+2)$, i.e., after GST for every v there exists a group S of honest nodes s.t.* $|S| \geq f+1$ *and* $\max_{\mathcal{P}_i \in S}\left\{t^{prop}_{\mathcal{P}_i,v}\right\} - \min_{\mathcal{P}_j \in S}\left\{t^{prop}_{\mathcal{P}_j,v}\right\} \leq 2\delta(f+2)$.

PROOF. Let $\mathcal{P}_i$ be the first node that entered view v at time t. $\mathcal{P}_i$ entered v since it received " QC, v" from Leader(r) and $v \leq r \leq v+f+1$ (Line 13). If Leader(r) is honest then we are done, since Leader(r) multicasted " QC, v" to all honest nodes (Line 24), and within δ all honest nodes will also enter view v by $t+\delta$.

Next, if Leader(r) is Byzantine, then it might have sent " QC, v" to a subset of the honest nodes, potentially only to $\mathcal{P}_i$. In order to form a " QC, v" , Leader(r) had to receive $2f+1$ " VOTE, v" messages (Line 23), meaning that at least $f+1$ honest nodes sent " VOTE, v" to Leader(r). Denote S as the group of those $f+1$ honest nodes.

Each node in S sent " VOTE, v" message since it received " TC, v" from Leader(r') for $v \leq r' \leq v+f+1$ (Line 7). Note that different nodes in S might have received " TC, v" from a different leader, i.e., Leader(r') might not be the same leader for each node in S.

After a node in S sent " VOTE, v" it will either receive a " QC, v" within 2δ and enter view v, or timeout after 2δ and send " VOTE, v" with " TC, v" to Leader$(v+1)$ (Line 19). They will continue to do so when not receiving " QC, v" for the next $f+1$ views after v. This ensures that at least one honest leader will receive " TC, v" after at most $t+2\delta f+\delta$. Then, this honest leader will multicast the " TC, v" it received (Line 21) and at most by $t+2\delta(f+1)$, all the honest nodes will receive " TC, v" . The honest nodes will then send " VOTE, v" to the honest leader, which will be able to create " QC, v" and multicast it. The " QC, v" will thus be received by all the honest nodes by $t+2\delta(f+2)$ and we are done. □

Next, we show that during the execution, an honest node will enter some new view.

CLAIM 3. *After GST, some honest node $\mathcal{P}_i$ enters a new view.*

PROOF. From Claim 2, if an honest node enters some view v, the time by which at least another f other honest nodes also enter v is bounded. Eventually, those honest nodes will timeout and $\mathsf{wish\_to\_advance}()$ will be invoked (Line 5), which will cause them to send " $\mathsf{WISH}, v+1$" to Leader$(v+1)$.

If Leader$(v+1)$ is honest, then it will send a " $\mathsf{TC}, v+1$" to all the nodes (Line 21) which will be followed by the leader sending a " $\mathsf{QC}, v+1$" (Line 23), and all honest nodes will enter view $v+1$.

If Leader$(v+1)$ is not honest then the protocol dictates that the honest nodes that wished to enter $v+1$ will continue to forward their " $\mathsf{WISH}, v+1$" message to the next leaders (up to Leader$(v+f+1)$, Line 10) until each of them receives " $\mathsf{TC}, v+1$" . This is guaranteed since at least one of those $f+1$ leaders is honest.

The same process is then followed for " $\mathsf{QC}, v+1$" (Line 18), and eventually all of those $v+1$ honest nodes will enter view $v+1$. □

LEMMA 4.1. *Cogsworth achieves eventual view synchronization (Property 1).*

PROOF. From Claim 3 an honest node eventually will enter a new view, and by Claim 2 at least $f+1$ honest nodes will enter the same view within a bounded time. By applying Claim 3 recursively and again, eventually, a view with an honest leader is reached and by Claim 1 all honest nodes will enter the view within 4δ.

Thus, for any $c \geq 0$, if the Cogsworth protocol is run with $\alpha = 4\delta + c$ it is guaranteed that all honest nodes will eventually execute the same view for $|\mathcal{I}| = c$.

The above arguments can be applied inductively, i.e., there exists an infinite number of such intervals and views in which view synchronization is reached, also ensuring that the views that synchronized also have an honest leader. □

LEMMA 4.2. *Cogsworth achieves synchronization validity (Property 2).*

PROOF. To enter a new view v a " QC, v" is needed, which is consisted of $2f+1$ " VOTE, v" messages i.e., at least $f+1$ are from honest nodes. An honest node will send " VOTE, v" message only when it receives a " TC, v" message, that requires $f+1$ " WISH, v" message, meaning at least one of those messages came from an honest node.

An honest node will send " WISH, v" when the upper-layer protocol invokes $\mathsf{wish\_to\_advance}()$ while it was in view $v-1$. □

This concludes the proof that Cogsworth is a synchronizer for any $\alpha \geq 4\delta$. Similar to the broadcast-based synchronizer, it allows upper-layer protocols to determine the time they spend in each view.

Latency and communication. Let v_{max}^{GST} be the maximum view an honest node is in at GST, and let X denote the number of consecutive Byzantine leaders after v_{max}^{GST}. Assuming that leaders are randomly allocated to views, then X is a random variable of a geometrical distribution with a mean of $n/(n-f)$. This means that in the worst case of $t = f = \lfloor n/3 \rfloor$, then $\mathbb{E}(X) = (3f+1)/(2f+1) \approx 3/2$.

Since when $f+1$ honest nodes at view v want to advance to view $v+1$, and if Leader$(v+1)$ is honest, all honest nodes enter view $v+1$ in constant time (Claim 1), the latency for view synchronization, in general, is $O(X \cdot \delta)$. For the same reasoning, this is also the case for any two intervals between view synchronizations (see Theorem 3.1).

In the worst-case of $X = t$, where t is the number of actual failures during the run, then latency is linear in the view duration, i.e., $O(t \cdot \delta)$. But, in the expected case of a constant number of consecutive Byzantine leaders after v_{max}^{GST}, the expected latency is $O(\delta)$.

For communication complexity, there is a difference between Byzantine failures and benign ones. If a Byzantine leader of a view r obtains "$\texttt{TC}, v$" for $r-(f+1) \leq v \leq r$, then it can forward the "$\texttt{TC}, v$" to all the $f+1$ leaders that follow view v and those leaders will multicast the message (Line 21), leading to expected $O(n^2)$ communication complexity, in the case of at least one Byzantine leader after v_{max}^{GST}. In the worst-case of a cascade of t failures after v_{max}^{GST}, the communication complexity is $O(t \cdot n^2)$.

In the case of benign failures, communication complexity is dependent on X, since the first correct leader after v_{max}^{GST} will get all nodes to enter his view and achieve view synchronization, and the benign leaders before it will only cause delays in terms of latency, but will not increase the overall number of messages sent. Thus, in general, the communication complexity with benign failures is $O(X \cdot n)$. In the worst-case of $X = t$ communication complexity is $O(t \cdot n)$, but in the average case it is linear, i.e., $O(n)$. For the same reasoning, this is also the case between any consecutive occurrences of view synchronization (see Theorem 3.2).

To sum-up, the *expected* latency for both *benign* and *Byzantine* failures is $O(\delta)$, and *worst-case* $O(t \cdot \delta)$. Communication complexity for *Byzantine* nodes is *optimistically* $O(n)$, *expected* $O(n^2)$ and *worst-case* $O(t \cdot n^2)$ and for benign failures is *expected* $O(n)$ and *worst-case* $O(t \cdot n)$.

Discussion. Cogsworth achieves expected constant latency and linear communication under a broad set of assumptions. It is another step in the direction of reaching the quadratic communication lower bound of Byzantine consensus in an asynchronous model [Dolev and Reischuk 1985].

In addition to Cogsworth we present in Appendix A two more view synchronization algorithms. The first one is view doubling, where nodes simply double their view duration when entering a new view, which guarantees that eventually all nodes will be in the same view for sufficiently long. The other algorithm is borrowed from consensus protocols such as PBFT [Castro et al. 1999] and SBFT [Gueta et al. 2019]. In Appendix A.3 we present a comprehensive discussion on all three algorithms.

5. USAGES AND IMPLEMENTATIONS OF SYNCHRONIZERS

In this section, we describe real-world usages of the view synchronization algorithms. First, it is worth mentioning that many times the terms "phase", "round", and "view" are mixed in different works. In this work when "view" is mentioned, the meaning is that all the nodes agree on some integer value, mapped to a specific node that acts as the leader.

There are SMR protocols where as long as the leader is driving progress in the protocol it is not changed. This will correspond to all the nodes staying in the same view, and this view can be divided into many phases. E.g., in PBFT [Castro et al. 1999] a single-shot consensus consists of two phases.

In an SMR protocol based on PBFT a view can consist of many more phases, all with the same leader as long as progress is made, and there is no bound on the view duration.

As mentioned in §1.2, in HotStuff [Yin et al. 2019], the view synchronization logic is encapsulated in a module named a PaceMaker, but does not provide a formal definition of what the PaceMaker does, nor an implementation. The most developed work which adopted HotStuff as the core of its consensus protocol is LibraBFT [The LibraBFT Team 2019]. In LibraBFT, a module also named a PaceMaker is in charge of advancing views. In this module whenever a node timeouts of its current view, say view v, it sends a message named "TimeoutMsg, v", and whenever it receives $2f+1$ of these messages, it advances to view v. In addition, the node sends an aggregated signature of these messages to the leader of view v, which according to the paper, if the leader of v is honest, guarantees that all other nodes will enter view v withing 2δ. The current implementation of the PaceMaker is linear communication as long as there are honest leaders, but quadratic upon reaching a view with a Byzantine one. The latency is constant.

Many other works on consensus rely on view synchronization as part of their design. For example, in [Gupta et al. 2019] a doubling view synchronization technique is used: "For the view-change process, each replica will start with a timeout δ and double this timeout after each view-change (exponential backoff). When communication becomes reliable, exponential backoff guarantees that all replicas will eventually view-change to the same view at the same time."

6. RELATED WORK

View synchronization in consensus protocols. The idea of doubling round duration to cope with partial synchrony borrows from the DLS work [Dwork et al. 1988], and has been employed in PBFT [Castro et al. 1999] and in various works based on DLS/PBFT [The LibraBFT Team 2019; Buchman et al. 2018; Yin et al. 2019]. In these works, nodes double the length of each view when no progress is made. The broadcast-based synchronization algorithm is also employed as part of the consensus protocol in works such as PBFT.

HotStuff [Yin et al. 2019] encapsulates view synchronization in a separate module named a PaceMaker. Here, we provide a formal definition, concrete solutions, and performance analysis of such a module. HotStuff is the core consensus protocol of various works such as Cypherium [Cypherium 2019], PaLa [Chan et al. 2018] and LibraBFT [The LibraBFT Team 2019]. Other consensus protocols such as Tendermint [Buchman et al. 2018] and Casper [Buterin and Griffith 2017] reported issues related to the liveness of their design [Milosevic 2018; Pyrofex Corporation 2018].

Notion of time in distributed systems. Causal ordering is a notion designed to give partial ordering to events in a distributed system. The most known protocols to provide such ordering are Lamport Timestamps [Lamport 1978] and vector clocks [Fidge 1988]. Both works assume a non-crash setting.

Another line of work stemmed from Awerbuch work on synchronizers [Awerbuch 1985]. The synchronizer in Awerbuch's work is designed to allow an algorithm that is designed to run in a synchronous network to run in an asynchronous network without any changes to the synchronous protocol itself. This work is orthogonal to the work in this paper.

Recently, Ford published preliminary work on Threshold Logical Clocks (TLC) [Ford 2019]. In a crash-fail asynchronous setting, TLC places a barrier on view advancement, i.e., nodes advance to view $v+1$ only after a threshold of them reached view v. A few techniques are also described on how to convert TLCs to work in the presence of Byzantine nodes. The TLC notion of a view "barrier" is orthogonal to view synchronization, though a 2-phase TLC is very similar to our reliable broadcast synchronizer.

Failure detectors. The seminal work of Chandra and Toueg [Chandra et al. 1996], [Chandra and Toueg 1996] introduces the leader election abstraction, denoted Ω, and prove it is the weakest failure detector needed to solve consensus. By using Ω, consensus protocols can usually be written in a more natural way. The view synchronization problem is similar to Ω, but differs in several ways. First, it lacks any notion of leader and isolates the view synchronization component. Second, view synchronization adds recurrence to the problem definition. Third, it has a built-in notion of view-duration: nodes commit to spend a constant tine in a view before moving to the next. Last, this paper focuses on latency and communication costs of synchronizer implementations.

Latency and message communication for consensus. Dutta et al. [Dutta et al. 2007] look at the number of rounds it takes to reach consensus in the crash-fail model after a time defined as GSR (Global Stabilization Round) which only correct nodes enter. This work provides an upper and a lower bound for reaching consensus in this setting. Other works such as [Alistarh et al. 2008; Dutta et al. 2005] further discuss the latency for reaching consensus in the crash-fail model. These works focus on the latency for reaching consensus after GST. Both bounds are tangential to our performance measures, as they analyze round latency.

Dolev et al. [Dolev and Reischuk 1985] showed a quadratic lower bound on communication complexity to reach Byzantine broadcast, which can be reduced to consensus. This lower bound is an intuitive baseline for work like ours, though it remains open to prove a quadratic lower bound on view synchronization per se.

Clock synchronization. The clock synchronization problem [Lamport and Melliar-Smith 1985] in a distributed system requires that the maximum difference between the local clock of the participating nodes is bounded throughout the execution, which is possible since most works assume a synchronous setting. The clock synchronization problem is well-defined and well-treaded, and there are many different algorithms to ensure this in different models, e.g., [Cristian 1989; Kopetz and Ochsenreiter 1987; Srikanth and Toueg 1987]. In practical distributed networks, the most prevalent protocol is NTP [Mills 1991]. Again, clock synchronization is an orthogonal notion to view synchronization, the latter guaranteeing to and stay in a view within a bounded window, but does not place any bound on the views of different nodes at any point in time.

7. CONCLUSION

We formally defined the *Byzantine view synchronization* problem, which bridges classic works on failure detectors aimed to solve one-time consensus, and SMR which consists of multiple one-time consensus instances. We presented Cogsworth which is a view synchronization algorithm that displays linear communication cost and constant latency under a broad variety of scenarios.

REFERENCES

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 (PODC '19)*. ACM, New York, NY, USA, 337–346. DOI: http://dx.doi.org/10.1145/3293611.3331612

Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. 2008. How to solve consensus in the smallest window of synchrony. In *International Symposium on Distributed Computing*. Springer, 32–46.

Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, and others. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.

Baruch Awerbuch. 1985. Complexity of network synchronization. *Journal of the ACM (JACM)* 32, 4 (1985), 804–823.

Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 514–532.

Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

Ethan Buchman. 2017. Tendermint: Byzantine fault tolerance in the age of blockchains (2016). (2017).

Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938* (2018).

Michael Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. 335–350. http://www.usenix.org/events/osdi06/tech/burrows.html

Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).

Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.

Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* (2014), 13.

Miguel Castro, Barbara Liskov, and others. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

T-H Hubert Chan, Rafael Pass, and Elaine Shi. 2018. PaLa: A Simple Partially Synchronous Blockchain. *IACR Cryptology ePrint Archive* 2018 (2018), 981.

Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)* 43, 4 (1996), 685–722.

Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8:1–8:22. https://dl.acm.org/citation.cfm?id=2491245

Flaviu Cristian. 1989. Probabilistic clock synchronization. *Distributed computing* 3, 3 (1989), 146–158.

Cypherium. 2019. CypherBFT: Enabling Decentralization for HotStuff. (2019). https://medium.com/@cypherium/cypherbft-enabling-decentralization-for-hotstuff-1713da26628b

Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)* 32, 1 (1985), 191–204.

Partha Dutta, Rachid Guerraoui, and Idit Keidar. 2007. The overhead of consensus failure recovery. *Distributed Computing* 19, 5-6 (2007), 373–386.

Partha Dutta, Rachid Guerraoui, and Leslie Lamport. 2005. How fast can eventual synchrony lead to consensus?. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 22–27.

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

etcd community. 2019. etcd. *https://etcd.io/* (2019).

C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf

Michael J Fischer, Nancy A Lynch, and Michael Merritt. 1986. Easy impossibility proofs for distributed consensus problems. *Distributed Computing* 1, 1 (1986), 26–39.

Bryan Ford. 2019. Threshold Logical Clocks for Asynchronous Distributed Coordination and Consensus. *CoRR* abs/1907.07010 (2019). http://arxiv.org/abs/1907.07010

Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 568–580.

Sky Guo. 2017. Cypherium: A Scalable and Permissionless Smart Contract Platform. *Draft v1.1 http://cypherium.wpengine.com/wp-content/uploads/2018/10/cypherium-whitepaper.pdf* (2017).

Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. *arXiv preprint arXiv:1911.00838* (2019).

Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

Sep Kamvar, Marek Olszewski, and Rene Reinsberg. 2019. Celo: A Multi-Asset Cryptographic Protocol for Decentralized Social Payments. *DRAFT version 0.24 https://storage.googleapis.com/celo_whitepapers/Celo__A_Multi_Asset_Cryptographic_Protocol_for_Decentralized_Social_Payments.pdf* (2019).

Hermann Kopetz and Wilhelm Ochsenreiter. 1987. Clock synchronization in distributed real-time systems. *IEEE Trans. Comput.* 100, 8 (1987), 933–940.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

Leslie Lamport. 1983. The weak Byzantine generals problem. *Journal of the ACM (JACM)* 30, 3 (1983), 668–676.

Leslie Lamport and others. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

Leslie Lamport and P Michael Melliar-Smith. 1985. Synchronizing clocks in the presence of faults. *Journal of the ACM (JACM)* 32, 1 (1985), 52–78.

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.

Libra Association Members. 2019. An Introduction to Libra. (2019). https://libra.org/en-US/wp-content/uploads/sites/23/2019/06/LibraWhitePaper_en_US.pdf

David L Mills. 1991. Internet time synchronization: the network time protocol. *IEEE Transactions on communications* 39, 10 (1991), 1482–1493.

Zarko Milosevic. 2018. Receiving 2f+1 Prevotes before Proposal might affect termination. (2018). https://github.com/tendermint/tendermint/issues/1745

Brian M Oki and Barbara H Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM, 8–17.

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

Pyrofex Corporation. 2018. Incorrect by Construction - CBC Casper Isn't Live. (2018). https://pyrofex.io/wp-content/uploads/2018/12/Incorrect-By-Construction.pdf

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

Victor Shoup. 2000. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 207–220.

TK Srikanth and Sam Toueg. 1987. Optimal clock synchronization. *Journal of the ACM (JACM)* 34, 3 (1987), 626–645.

The LibraBFT Team. 2019. State machine replication in the Libra Blockchain, version 3. (2019). https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf

VMware. 2019. Project Concord BFT. (2019). https://vmware.github.io/concord-bft/

Maofan Yin, Dahlia Malkhi, MK Reiter and, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC'19).*

Algorithm 2: View doubling synchronizer for node $\mathcal{P}$

```
initialize :
    $wish \leftarrow 0$
    $curr \leftarrow 0$
    $view\_duration \leftarrow \beta$ /* $\beta$ is a predefined value for the duration of the first view */
on wish_to_advance():
    $wish \leftarrow wish + 1$
after $view\_duration$ has passed since last changing its value:
    $curr \leftarrow curr + 1$
    $view\_duration \leftarrow 2 \times view\_duration$
    if $wish \geq curr$ then
        propose_view($curr$)
```

A. PROTOCOLS FOR VIEW SYNCHRONIZATION

In this section we place into the view synchronization framework two view synchronization algorithms which are used in various consensus protocols, and prove their correctness, as well as discuss their latency and message complexity.

All protocol messages between nodes are signed and verified; for brevity, we omit the details about the cryptographic signatures.

A.1. View Doubling Synchronizer

A.1.1. Overview. A solution approach inspired by PBFT [Castro et al. 1999] is to use view doubling as the view synchronization technique. In this approach, each view has a timer, and if no progress is made the node tries to move to the next view and doubles the timer time for the next view. Whenever progress is made, the node resets its timer. This approach is intertwined with the consensus protocol itself, making it hard to separate, as the messages of the consensus protocol are part of the mechanism used to reset the timer.

We adopt this approach and turn it into an independent synchronizer that requires no messages. Fist, the nodes need to agree on some predefined constant $\beta > 0$ which is the duration of the first view. Next, there exists some global view duration mapping $VD(\cdot) : \mathbb{N} \mapsto \mathbb{R}^+$, which maps a view v to its duration: $VD(v) = 2^v\beta$. A node in a certain view must move to the next view once this duration passes, regardless of the outer protocol actions.

The view doubling protocol is described in Alg. 2. A node starts at view 0 (Line 3) and a view duration of $\beta > 0$ (Line 4). Next, when wish_to_advance() is called, a counter named *wish* is incremented (Line 5). This counter guarantees validity by moving to a view v only when the *wish* counter reaches v. Every time a view ends (Line 7), an internal counter *curr* is incremented, and if the *wish* allows it, the synchronizer outputs propose_view(v) with a new view v.

A.1.2. Correctness. We show that the view doubling protocol achieves the properties required by a synchronizer.

LEMMA A.1. *The view doubling protocol achieves view synchronization (Property 1).*

PROOF. Since this protocol does not require sending messages between nodes, the Byzantine nodes cannot affect the behavior of the honest nodes, and we can treat all nodes as honest.

Recall that $t = 0$ denotes the time by which all the honest nodes started their local execution of Alg. 2. Let $init_i$ be the view at which node $\mathcal{P}_i$ is at during $t = 0$. W.l.o.g assume $init_1 \leq init_2 \leq \cdots \leq init_n$ at time $t = 0$. It follows from the definition of $init_i$ and the sum of a geometric series that

$$t^{prop}_{\mathcal{P}_i,v} = \beta\left(2^v - 2^{init_i}\right). \tag{1}$$

We begin by showing that for every $i \leq j$ the following condition holds: $t^{prop}_{\mathcal{P}_i,v} \geq t^{prop}_{\mathcal{P}_j,v}$ for any view v. Let $k = init_i$ and $l = init_j$. From the ordering of the node starting times, for all $k \leq l$. We get:

$$t^{prop}_{\mathcal{P}_i,v} \geq t^{prop}_{\mathcal{P}_j,v} \Leftrightarrow \beta\left(2^v - 2^k\right) \geq \beta\left(2^v - 2^l\right) \Leftrightarrow l \geq k.$$

Hence, for $i \leq j$, since at $t = 0$ node $\mathcal{P}_j$ had a view number larger than $\mathcal{P}_i$, then $\mathcal{P}_j$ will start all future views before $\mathcal{P}_i$.

Next, let $k = init_1$ and $l = init_n$, i.e., the minimal view and the maximal view at $t = 0$ respectively. To prove that the first interval of view synchronization is achieved, it suffices to show that for any constant $c \geq 0$ there exists a time interval $\mathcal{I}$ and a view v such that $|\mathcal{I}| \geq c$ and $t^{prop}_{n,v+1} - t^{prop}_{1,v} \geq |\mathcal{I}|$. Using this, we will show that there exists an infinite number of such intervals and views that will conclude the proof. This also ensures that there is an infinite number of such views with honest leaders.

Indeed, first note that as shown above, node $\mathcal{P}_n$ will start view v before any other node in the system. The left-hand side of the equation is the time length in which both node $\mathcal{P}_n$ and node $\mathcal{P}_1$ execute together view v. If the left-hand side is negative, then there does not exist an overlap, and if it is positive then an overlap exists.

We get

$$t^{prop}_{n,v+1} - t^{prop}_{1,v} \geq |\mathcal{I}| \Leftrightarrow \beta\left(2^{v+1} - 2^l\right) - \beta\left(2^v - 2^k\right) \geq |\mathcal{I}| \Leftrightarrow \beta\left[2^v + \left(2^k - 2^l\right)\right] \geq |\mathcal{I}|. \quad (2)$$

For any $c \geq 0$ there exists a minimum view number v' such that the inequality holds, and since k is the minimum view number at $t = 0$ this solution holds for any other node $\mathcal{P}_i$ as well. In addition, for any $v \geq v'$ the inequality also holds, meaning there is an infinite number of solutions for it, including an infinite number of views with an honest leader.

If wish_to_advance() is called in intervals with $0 < \alpha \leq \beta$ then by the time the value of *curr* reaches some view value v, *wish* will always be bigger than *curr*, meaning the condition in Line 10 will always be true, and the synchronizer will always propose view v by the time stated in Eq. (1). □

LEMMA A.2. *The view doubling protocol achieves synchronization validity (Property 2).*

PROOF. The if condition in Line 10 ensures that the output of the synchronizer will always be a view that a node wished to advance to. □

This concludes the proof that view doubling is a synchronizer for any $0 < \alpha \leq \beta$.

A.1.3. Latency and communication. Since the protocol sends no messages between the nodes, it is immediate that the communication complexity is 0.

As for latency, the minimal v^* satisfying Eq. (2) grows with $c\left(2^{init_n} - 2^{init_1}\right)$. Since the initial view-gap $init_n - init_1$ is unbounded, so is the view v^* in which synchronization is reached. The latency to synchronization is $t^{prop}_{\mathcal{P}_1,v^*} = 2^{v^*} - 2^{init_1}$, also unbounded.

A.2. Broadcast-Based Synchronizer

A.2.1. Overview. Another leaderless approach is based on the Bracha reliable broadcast protocol [Bracha 1987] and is presented in Alg. 3. In this protocol, when a node wants to advance to the next view v it multicasts a "WISH, v" message (*multicast* means to send the message to all the nodes including the sender) (Line 3). When at least $f + 1$ "WISH, v" messages are received by an honest node, it multicasts "WISH, v" as well (Line 5). A node advances to view v upon receiving $2f + 1$ "WISH, v" messages (Line 7).

A.2.2. Correctness. We start by showing that the broadcast-based synchronizer achieves eventual view synchronization (Property 1) for any $\alpha \geq 2\delta$. Thus, the claims and lemmas below assume this.

Algorithm 3: Broadcast-based synchronizer for node $\mathcal{P}$

```
initialize :
    $curr \leftarrow 0$
on wish_to_advance():
    multicast "WISH, $curr+1$" to all nodes (including self)
upon receiving $f+1$ "WISH, $v$" messages and not sending "WISH, $v$" before:
    multicast "WISH, $v$" to all nodes (including self)
upon receiving $2f{+}1$ "WISH, $v$" messages and $v > curr$
    $curr \leftarrow v$
    propose_view($v$)
```

CLAIM 4. *After GST, whenever an honest node enters view v at time t, all other honest nodes enter view v by $t+2\delta$, i.e.,* $\max_{\mathcal{P}_i \in H}\left\{t^{prop}_{\mathcal{P}_i,v}\right\} - \min_{\mathcal{P}_j \in H}\left\{t^{prop}_{\mathcal{P}_j,v}\right\} \leq 2\delta$.

PROOF. Suppose an honest node $\mathcal{P}_i \in H$ enters view v at time $t^{prop}_{\mathcal{P}_i,v} = t$, then it received $2f+1$ " WISH, v" messages, from at least $f+1$ honest nodes (Line 7).

Since the only option for an honest node to disseminate " WISH, v" message is by multicasting it, then by $t+\delta$ all nodes will receive at least $f+1$ " WISH, v" messages. Then, any left honest nodes (at most f nodes) will thus receive enough " WISH, v" to multicast the message on their own (Line 5) which will be received by $t+2\delta$ by all the nodes. This ensures that all the honest nodes receive $2f+1$ " WISH, v" messages and enter view v by $t+2\delta$. □

CLAIM 5. *After GST, eventually an honest node $\mathcal{P}_i$ enters some new view.*

PROOF. All honest nodes begin their local execution at view 0, potentially at different times. Based on the protocol eventually at least $f+1$ nodes (some of them might be Byzantine) send " WISH, 1". This is because wish_to_advance() is called every α. Thus, eventually all honest nodes will reach view 1, and from Claim 4 the difference between their entry is at most 2δ after GST.

The above argument can be applied inductively. Suppose at time t node $\mathcal{P}_i$ is at view v. We again know that by $t+2\delta$ all other honest nodes are also at view v, and once $f+1$ " WISH, $v+1$" are sent all honest nodes will eventually enter view $v+1$, and we are done. □

LEMMA A.3. *The broadcast-based protocol achieves view synchronization (Property 1).*

PROOF. From Claim 5 an honest node will eventually advance to some new view v and from Claim 4 after 2δ all other honest nodes will join it. For any $c \geq 0$, if the honest nodes call wish_to_advance() every $\alpha = 2\delta + c$ then it is guaranteed that all the honest nodes will execute view v together for at least $|\mathcal{I}| = c$ time, since it requires $f+1$ messages to move to view $v+1$, i.e., at least one message is sent from an honest node.

This argument can be applied inductively, and each view after GST is synchronized, thus making an infinite number of time intervals and views which all honest leaders execute at the same time. □

LEMMA A.4. *The broadcast-based synchronizer achieves synchronization validity (Property 2).*

PROOF. In order for an honest node to advance to view v it has to receive $2f+1$ " WISH, v" messages (Line 7). From those, at least $f+1$ originated from honest nodes. An honest node can send " WISH, v" on two scenarios:

(i) wish_to_advance() was called when the node was at view $v-1$ (Line 3) and we are done.

(ii) It received $f+1$ " WISH, v" messages (Line 5), meaning at least one honest node which already sent the message was at view $v-1$ and called wish_to_advance() and again we are done. □

This concludes the proof that the broadcast-based synchronizer is a view synchronizer for any $\alpha \geq 2\delta$.

A.2.3. Latency and communication. The broadcast-based algorithm synchronizes every view after GST within 2δ. Since the leaders of each view are allocated by the mapping $\text{Leader}(\cdot)$, in expectation every $\approx 3/2$ nodes have an honest leader (see the communication complexity analysis done for Cogsworth in §4). Therefore, for latency, the broadcast-based synchronizer will take an expected constant time to reach view synchronization after GST, as we have proved, and also the same between every two consecutive occurrences of view synchronization. Thus, the latency of this protocol is expected $O(\delta)$. In the worst-case of t consecutive failures, the latency is $O(t \cdot \delta)$.

For communication costs, the protocol requires that every node sends one " WISH, v" message to all the other nodes, and since the latency is expected constant, the overall communication costs are also expected quadratic, i.e., $O(n^2)$. In the worst-case of t consecutive failures, the communication complexity is $O(t \cdot n^2)$.

A.3. Discussion

The three presented synchronizers in the paper have tradeoffs in their latency and communication costs, which are summarized in Table I. Hence, a protocol designer may choose a synchronizer based on its needs and constraints. It may be possible to create combinations of the three protocols and achieve hybrid characteristics; we leave such variations for future work.

In addition, there are differences in the constraints on the parameter α in these protocols, which is the time interval between two successive calls to wish_to_advance() (see Property 1). The view doubling synchronizer prescribes a precise α, which results in each view duration to be exactly twice as its predecessor. In the other two synchronizers there is only a lower bound on α: in the broadcast-based it is 2δ, and in Cogsworth it is 4δ.

This difference is significant. Suppose an upper-layer protocol utilizing the synchronizer wishes to spend an unbounded amount of time in each view as long as progress is made, and triggers a view-change upon detecting that progress is lost. While the broadcast-based and Cogsworth algorithms allow this upper-layer behavior, the view doubling technique does not, and thus may influence the decision on which view synchronization algorithm to choose.