Lumiere: Making Optimal BFT for Partial Synchrony Practical

Andrew Lewis-Pye
London School of Economics
United Kingdom
a.lewis7@lse.ac.uk

Dahlia Malkhi
UC Santa Barbara
USA
dahliamalkhi@ucsb.edu

Oded Naor
StarkWare
Israel
odednaor@gmail.com

Kartik Nayak
Duke University
USA
kartik@cs.duke.edu

ABSTRACT

The *view synchronization* problem lies at the heart of many Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) protocols in the partial synchrony model, since these protocols are usually based on *views*. Liveness is guaranteed if honest processors spend a sufficiently long time in the same view during periods of synchrony, and if the leader of the view is honest. Ensuring that these conditions occur, known as *Byzantine View Synchronization (BVS)*, has turned out to be the performance bottleneck of many BFT SMR protocols.

A recent line of work [7, 12] has shown that, by using an appropriate view synchronization protocol, BFT SMR protocols can achieve $O(n^2)$ communication complexity in the worst case after GST, thereby finally matching the lower bound established by Dolev and Reischuk in 1985 [9]. However, these protocols suffer from two major issues, hampering praticality:

(i) When implemented so as to be *optimistically responsive*, even a single Byzantine processor may infinitely often cause $\Omega(n\Delta)$ latency between consecutive consensus decisions.
(ii) Even in the absence of Byzantine action, infinitely many views require honest processors to send $\Omega(n^2)$ messages.

Here, we present Lumiere, an optimistically responsive BVS protocol which maintains optimal worst-case communication complexity while simultaneously addressing the two issues above: for the first time, Lumiere enables BFT consensus solutions in the partial synchrony setting that have $O(n^2)$ worst-case communication complexity, and that eventually always (i.e., except for a small constant number of "warmup" decisions) have communication complexity and latency which is linear in the number of actual faults in the execution.

CCS CONCEPTS

• **Theory of computation** → **Distributed algorithms**.

PODC '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0668-4/24/06.
https://doi.org/10.1145/3662158.3662787

KEYWORDS

Distributed Systems, Byzantine View Synchronization

ACM Reference Format:
Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. 2024. Lumiere: Making Optimal BFT for Partial Synchrony Practical. In *ACM Symposium on Principles of Distributed Computing (PODC '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3662158.3662787

1 INTRODUCTION

State machine replication (SMR) is a central topic in distributed computing and refers to the task of implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas [18]. Driven partly by high levels of investment in 'blockchain' technology, recent years have seen interest in developing SMR protocols that work efficiently at scale [8]. In concrete terms, this means looking to minimize the latency and the communication complexity per consensus decision as a function of the number of processors (participants) n.

SMR protocols typically aim to achieve Byzantine fault tolerance, i.e., consensus among processors (server replicas) even if a bounded proportion of the processors are Byzantine and behave arbitrarily/maliciously. The partial synchrony model [10] is a common networking model on which many of these protocols are based, and this model can be seen as a practical compromise between the synchronous and asynchronous communication models. This model assumes a point in time called the global stabilisation time (GST) such that any message sent at time t must arrive by time $\max\{\text{GST}, t\} + \Delta$. While Δ is known, the value of GST is unknown to the protocol. *Optimal resiliency* in the partial synchrony communication model means tolerating up to f Byzantine processors among n processors, where f is the largest integer less than $n/3$ [10].

The view-based paradigm and view synchronization in partial synchrony. Many BFT SMR protocols [6, 11, 15, 20] in the partial synchrony model employ a *view-based* paradigm. The instructions for such protocols are divided into *views*, each view having a designated *leader* to drive progress. A consensus decision is guaranteed to be reached during periods of synchrony whenever honest processors spend a sufficiently long time together in any view with an honest leader. The problem of ensuring that processors synchronize for long enough in the same view is known as the *Byzantine View Synchronization (BVS)* problem.

HotStuff [20] was the first BFT SMR protocol to decouple the core consensus logic from a "Pacemaker" module that implements BVS, but left the pacemaker implementation unspecified. The core consensus logic in HotStuff requires quadratic communication complexity in the worst case and linear latency, while each view requires only linear complexity and a constant number of rounds. The task of view synchronization therefore becomes the efficiency bottleneck and the key question becomes: *Can we design a Byzantine view synchronization protocol with optimal communication complexity and latency?*

The desired communication complexity and latency requirements need some elaboration. Worst-case complexity should be $O(n^2)$ to meet the Dolev-Reischuk bound [9]. However, when there are fewer actual faults $f_a \leq f$, one would hope for the worst-case complexity between every pair of consensus decisions to be a function of f_a such that complexity is $o(n^2)$ when $f_a = o(n)$. Likewise, whereas worst-case latency is $O(n\Delta)$, one would hope for latency $o(n\Delta)$ in the face of a small number f_a of actual faults. Furthermore, given that the core consensus logic can provide *optimistic responsiveness*, such a property would be desirable in BVS also. Roughly, this means that the protocol should function at 'network speed' if it turns out that the actual number of faults $f_a \leq f$ is 0: if $f_a = 0$, the protocol should be live during periods when message delay is less than the given bound Δ, and latency should be a function of the actual (unknown) message delay δ. This is important because the actual message delay δ may be much smaller than Δ when the latter value is conservatively set to ensure liveness under a wide range of network conditions. More formally, we can say that a protocol is optimistically responsive if, subsequent to some finite time after GST, the worst-case latency between synchronized views with honest leaders (each of which will produce a consensus decision) is $O(\delta)$ in the case that $f_a = 0$. Generalizing this, a protocol is *smoothly optimistically responsive* if the corresponding bound is $O(\Delta f_a + \delta)$ for any value of $f_a \leq f$.

Recent works have addressed the Byzantine view synchronization problem and achieved some of the above described goals [3, 7, 12, 16, 17]: Cogsworth [16] is the first optimistically linear Byzantine view synchronization protocol against a constant number of benign failures, and NK20 [17] robustified it against constant Byzantine failures, but both have non-optimal worst-case; LP22 [12] and RareSync [7] require a large latency with fewer actual number of faults, and Fever [13] stipulates the first view is already synchronized which may be difficult to bootstrap in practice. Lumiere is the first Byzantine view synchronization protocol in the partial synchrony model to achieve all of these properties simultaneously. They key result is the following:

Theorem 1.1. *Lumiere is a BVS protocol for the partial synchrony model with the following properties:*

(1) Worst-case communication complexity $O(n^2)$.
(2) Worst-case latency $O(n\Delta)$.
(3) The protocol is smoothly optimistically responsive.
(4) Eventual worst case communication complexity $O(f_a n + n)$.

All terms described in Theorem 1.1 are formally defined in Section 2. Table 1 compares the relevant performance measures for state-of-the-art protocols. The comparison is described in more detail in Section 5.

We emphasize that Lumiere is the first BVS protocol for partial synchrony that satisfies (1) and (2) from the statement of Theorem 1.1 while also satisfying *either* (3) or (4) (and, in fact, satisfies both of these properties). The basic approach to describing a protocol that is smoothly optimistically responsive while satisfying properties (1) and (2) is to combine techniques from LP22 [12] and Fever [13]. LP22 achieves worst-case communication complexity $O(n^2)$ by dividing the instructions into *epochs*, where each epoch consists of $f + 1$ views. By performing a heavy ($\Theta(n^2)$ communication complexity) synchronization process only once at the beginning of each epoch, one can avoid the need for synchronization within epochs, thereby matching the Dolev-Reischuk bound [9]. LP22 also achieves optimistic responsiveness, but suffers from the issue that even a *single* faulty leader is then able to achieve $\Theta(n\Delta)$ delays. Borrowing a technique from Fever, views with correct leaders can advance at the speed of the network, rather than waiting for pre-set synchronized slots within the epoch. This removes the large latency of LP22 and thus, simultaneously achieve properties (1)–(3).

The most technically complex task is then to modify the protocol so as to also ensure that the eventual worst case communication complexity is $O(f_a n + n)$ (property (4)). To do so requires establishing conditions that allow processors to stop carrying out heavy epoch changes once sufficiently synchronized. This involves a number of substantial technical complexities that are discussed in depth in Section 3.5.

Due to space limitations, the full proof of correctness has been deferred to the online version of this paper [14].

2 THE SETUP

For simplicity, we assume $n = 3f + 1$ and consider a set $\Pi = \{p_1, \ldots, p_n\}$ of n processors. At most f processors may become corrupted by the adversary during the course of the execution, and may then display *Byzantine* (arbitrary) behaviour. We let f_a denote the actual number of processors that become corrupted. Processors that never become corrupted by the adversary are referred to as *honest*.

Cryptographic assumptions. Our cryptographic assumptions are standard for papers on this topic. Processors communicate by point-to-point authenticated channels. We use a cryptographic signature scheme, a public key infrastructure (PKI) to validate signatures, and a threshold signature scheme [1, 19]. The threshold signature scheme is used to create a compact signature of m-of-n processors, as in other consensus and view synchronisation protocols [20]. In this paper, either $m = f + 1$ or $m = 2f + 1$. The size of a threshold signature is $O(\kappa)$, where κ is a security parameter, and does not depend on m or n. We assume a computationally bounded adversary. Following a common standard in distributed computing and for simplicity of presentation (to avoid the analysis of negligible error probabilities), we assume these cryptographic schemes are perfect, i.e. we restrict attention to executions in which the adversary is unable to break these cryptographic schemes.

The partial synchrony model. As noted above, processors communicate using point-to-point authenticated channels. We consider the standard partial synchrony model, whereby a message sent at time t must arrive by time $\max\{\text{GST}, t\} + \Delta$. While Δ is known, the

Protocol	Cogsworth NK20	LP22	Fever	Lumiere (this work)
Model	Partial Synchrony	Partial Synchrony	Bounded Clocks	Partial Synchrony
Worst-case Communication	$O(n^3)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Eventual Worst-case Communication	$O(n + nf_a^2)$	$O(n^2)$	$O(nf_a + n)$	$O(nf_a + n)$
Worst-case Latency	$O(n^2\Delta)$	$O(n\Delta)$	$O(f_a\Delta + \delta)$	$O(n\Delta)$
Eventual Worst-case Latency	$O(f_a^2\Delta + \delta)$	$O(n\Delta)$	$O(f_a\Delta + \delta)$	$O(f_a\Delta + \delta)$

Table 1: Summary of the results for state-of-the-art optimistically responsive protocols.

value of GST is unknown to the protocol. The adversary chooses GST and also message delivery times, subject to the constraints already defined. We let δ denote the actual (unknown) upper bound on message delay after GST. Each processor p also maintains a local clock value $\texttt{lc}(p)$. We assume that each processor p may join the protocol with $\texttt{lc}(p) = 0$ at any arbitrary time prior to GST, and that processors may experience arbitrary clock drift prior to GST. For simplicity we assume that, for honest p after GST, $\texttt{lc}(p)$ advances in real time, except when p pauses $\texttt{lc}(p)$ or bumps it forward (according to the protocol instructions). However, our analysis is easily modified to deal with a scenario where local clocks have bounded drift during any interval after GST in which they are not paused or bumped forward. When we wish to make the dependence on t explicit, we write $\texttt{lc}(p, t)$ to denote the value $\texttt{lc}(p)$ at time t.

The underlying protocol. We suppose view synchronisation is required for some underlying protocol (such as Hotstuff) with the following properties:

- **Views**. Instructions are divided into views. Each view v has a designated *leader*, denoted $\texttt{lead}(v)$.
- **Quorum certificates**. The successful completion of a view v is marked by all processors receiving a *Quorum Certificate* (QC) for view v. The QC is a threshold signature of length $O(\kappa)$ (for the security parameter κ that determines the length of signatures and hash values) combining $2f + 1$ signatures from different processors testifying that they have completed the instructions for the view. In a chained implementation of Hotstuff, for example, the leader will propose a block, processors will send votes for the block to the leader, who will then combine those votes into a QC and send this to all processors. Alternatively, one could consider a (non-chained) implementation of Hotstuff, in which the relevant QC corresponds to a successful third round of voting.
- **Sufficient time for view completion**. We suppose:

($\diamond_1$) There exists some known $x \geq 2$ such that if $\texttt{lead}(v)$ is honest, if (the global time) $t \geq$ GST, and if at least $2f + 1$ honest processors are in view v from time t until either they receive a QC for view v or until $t + x\delta$, then all honest processors will receive a QC for view v by time $t + x\delta$, so long as all messages sent by honest processors while in view v are received within time $\delta \leq \Delta$.

($\diamond_2$) No view v produces a QC unless there is some non-zero interval of time during which at least $2f + 1$ processors all act as if honest and in view v.

The view synchronisation task. For x as above, we must ensure:

(1) If an honest processor is in view v at time t and in view v' at $t' \geq t$, then $v' \geq v$.
(2) There exists some honest $\texttt{lead}(v)$ and $t \geq$ GST such that each honest processor is in view v from time t until either it receives a QC for view v or until $t + x\Delta$.

Condition (1) above is required by standard view-based SMR protocols to ensure consistency. Since GST is unknown to the protocol, condition (2) suffices to ensure the successful completion of infinitely many views with honest leaders. By a BVS protocol, we mean a protocol which determines when processors enters views and which satisfies conditions (1) and (2) above.

Complexity measures. All messages sent by honest processors will be of length $O(\kappa)$, where κ is the security parameter determining the length of signatures and hash values. We make the following definitions. Given $T \geq$ GST, let t_T^* be the least time $> T$ at which, for some view v, the underlying protocol has honest $\texttt{lead}(v)$ produce a QC for view v (if there exists no such time, set $t_T^* := \infty$). Then:

- The *worst-case communication complexity after* T, denoted W_T, is the maximum number of messages sent by correct processors (combined) between time T and t_T^*.
- The *worst-case communication complexity* is the worst-case communication complexity after GST+Δ.
- The *eventual worst-case communication complexity* is $\limsup_{T \to \infty} W_T$.

- The worst-case latency is the maximum possible value of $t^*_{\text{GST}} - \text{GST}$.
- The eventual worst-case latency is $\limsup_{T\to\infty} t^*_T - T$.

We note that Lumiere achieves its eventual worst-case communication complexity and latency for T which is within expected $O(n\Delta)$ time of GST.

3 OVERVIEW OF LUMIERE

In this section, we give an informal overview of the Lumiere protocol. Since the protocol itself is quite simple and hence practical to implement, we start with a brief protocol synopsis in Section 3.1.

The insights behind Lumiere and its analysis are more involved. Therefore, following the synopsis, we explain the ingredients that make it work. First, in Section 3.2, we review the LP22 protocol, and explain why it suffers from the two weaknesses described in Section 1. Then, in Section 3.3, we review the basic idea behind the Fever protocol. In Section 3.4, we describe how to combine the techniques developed by LP22 and Fever so as to give a protocol which has $O(n^2)$ worst-case communication complexity and which is smoothly optimistically responsive. Finally, in Section 3.5, we describe how to remove the need for views with $\Omega(n^2)$ communication complexity within time $O(n\Delta)$ of GST.

3.1 Lumiere Synopsis

Borrowing from RareSync and LP22, Lumiere batches views into *epochs*, and intertwines two synchronization procedures: a "heavy" epoch synchronization procedure and a "light" non-epoch synchronization procedure (throughout the paper, we enumerate views and epochs starting from 0, so that the first view is view 0 and the first epoch is epoch 0).

More specifically, at the start of some epochs, Lumiere employs a two round all-to-all broadcast procedure whose quadratic communication cost is amortized over all the views in the epoch. Importantly, Lumiere introduces a new mechanism that prevents performing such epoch synchronizations after a *successful* epoch generating QCs by $2f + 1$ leaders. This guarantees that only an expected constant-bounded number of heavy synchronizations will occur after GST. This mechanism is explained in detail in Section 3.5.

Within each epoch, Lumiere employs a light view synchronization procedure, which entails linear message complexity per view, and which allows processors to 'bump' their clocks forward and begin the instructions for the next view when they receive a QC – this process of 'bumping' clocks is explained in detail in Sections 3.3 and 3.4. Bumping clocks in this way produces a protocol that is smoothly optimistically responsive.

The above is the entire protocol. However, to make this work we need to tune a parameter Γ of the protocol that determines the view timers, so as to guarantee two things.

Firstly, we need that:

(a) If the $f+1$ honest processors whose clocks are most advanced begin some view with an honest leader within time Γ of one another after GST, the leader can generate a QC.

Second, because epoch synchronizations stop after a successful epoch, we must also guarantee that, in that event, either the clocks of honest processors are already sufficiently synchronized, or else the production of QCs by honest leaders will reduce the gap between the clocks of honest processes so that they are sufficiently synchronized soon after. To this end, we need that:

(b) The generation of QCs by honest leaders after GST "shrinks" the gap between the clocks of the $f + 1$ honest processors whose clocks are most advanced (unless this gap is already less than Γ).

It turns out that both (a) and (b) are accomplished simply by tuning the parameter Γ, and we will discuss appropriate values for Γ in the sections that follow.

The protocol is described in detail in Sections 3.4, 3.5 and 4.

3.2 Overview of LP22

Epochs. The core idea behind LP22 revolves around the concept of epochs: For every e, the sequence $f+1$ views $[e(f+1), \ldots, e(f+1)+f]$ is referred to as *epoch* e. We define $V(e) := e \cdot (f+1)$ and $E(v) := \lfloor v/(f+1) \rfloor$, so that view $V(e)$ is the first view of epoch e and $E(v)$ is the epoch to which view v belongs. The first view of each epoch is also called an *epoch view*, and all other views are called *non-epoch* views. The leader of view v is processor v mod n.

The clock time corresponding to a view. Clock times are not really necessary for specifying the LP22 protocol (which could equally be presented using 'timers'), but we wish to give a presentation here which is as similar as possible to our presentation of Lumiere later on.

As explained in Section 2, each processor maintains a local clock value $\mathtt{lc}(p)$. We also consider a clock time corresponding to each view, denoted $\mathtt{c}_v := \Gamma v$. Here Γ should be thought of as the length of time allocated to view v, and (for LP22) can be set to $\Gamma := x\Delta + \Delta$ (where x is as specified in Section 2). Roughly, the idea is that the processor p wishes to execute the instructions for view v once its local clock reaches $\mathtt{c}_v$.

The instructions for entering epoch views. Let $v := V(e)$. Processor p wishes to enter epoch e and view v if it is presently in a lower view and once its local clock reaches $\mathtt{c}_v$. At this point, it pauses its local clock and sends an epoch `view` v message to all processors,[1] indicating that it wishes to enter epoch e. Upon receiving epoch `view` v messages from $2f + 1$ distinct processors while in a view $< v$, any honest processor combines these into a single threshold signature, which is called an Epoch Certificate (EC) for view v, and sends the EC to all processors. Upon seeing an EC for view v while in any lower view, any honest processor sets $\mathtt{lc}(p) := \mathtt{c}_v$, unpauses its local clock if paused, and then enters epoch e and view v. Note that this process involves honest processors sending $\Theta(n^2)$ messages (combined). If $t \geq \text{GST}$ is the first time at which an honest processor enters epoch e, then all honest processors see the corresponding EC giving them permission to enter epoch e by time $t + \Delta$.

The instructions for entering non-epoch views. If we did not require a protocol that is optimistically responsive, we could simply have each processor enter non-epoch view v when its local clock

[1] It is convenient throughout to assume that when a processor sends a message to all processors, this includes itself.

reaches $\mathtt{c}_v$. To achieve optimistic responsiveness, LP22 uses a simple trick. Processor p enters non-epoch view v when the first of the following events occurs:

- Its local clock reaches $\mathtt{c}_v$, or;
- Processor p sees a QC for view $v-1$.

High level analysis of the protocol. The key insight is that, while the process for entering epoch e entails honest processors sending $\Theta(n^2)$ messages (combined), no further message sending is then required to achieve synchronization within the epoch. If the first honest processor to enter epoch e does so after GST, then all honest processors will see the corresponding EC within time Δ of each other. Suppose v is the first view of epoch e with an honest leader. If no leader has already produced a QC for some view $< v$ in the epoch, then view v will produce a QC.

To see that the protocol is optimistically responsive, suppose all leaders of an epoch are honest and let δ be the actual upper bound on message delay after GST. If the first honest processor to enter epoch e does so after GST, then (according to the assumptions of Section 2) the first leader will produce a QC in time $O(\delta)$. All honest processors will receive this QC within time δ of each other, which means that the leader of the second view will then produce a QC in time $O(\delta)$, and so on.

It is also not difficult to see that the protocol suffers from the two issues (i) and (ii) outlined in Section 1. First of all, we have already noted that entering each epoch requires honest processors to send $\Theta(n^2)$ messages (combined), meaning that the eventual worst-case communication complexity is $\Theta(n^2)$. To see that even a single Byzantine processor can cause a latency of $O(n\Delta)$ between consensus decisions, consider what happens when the first f leaders of an epoch produce QCs very quickly. If the last leader of the epoch is Byzantine, then honest processors must then wait for time almost $(f+1)\Gamma$ before wishing to enter the next epoch. Figure 1 illustrates a scenario with three good views producing QCs quickly, a faulty fourth view, and the fifth view suffering almost a 3Γ delay.

3.3 Overview of Fever

As noted previously, Fever makes stronger assumptions regarding *clock synchronization* than are standard (the assumptions of this paper are standard and are outlined in Section 2). However, we will show that the techniques developed by Fever can be combined with the LP22 protocol to give a protocol that is smoothly optimistically responsive. The fundamental idea behind Fever stems from consideration of what we refer to as the *honest gap*:

*Definition 3.1 (**Defining the honest gaps**).* At any time t, we let $p_{i,t}$ be the honest processor whose local clock is the i^{th} most advanced, breaking ties arbitrarily. So, $p_{1,t}$ is the honest processor whose local clock is most advanced. For $i \in [1, 2f+1]$, we define the i^{th} *honest gap* at time t to be $\mathtt{hg}_{i,t} := \mathtt{lc}(p_{1,t}, t) - \mathtt{lc}(p_{i,t}, t)$. In particular, $\mathtt{hg}_{f+1,t}$ is the gap between the local clock of the most advanced honest processor and the local clock of the $(f+1)^{\text{st}}$ most advanced honest processor.

Recall that, in Section 3.2, Γ was the maximum length of time allotted to each view, and that we set $\Gamma := (x+1)\Delta$ (where x is as specified in Section 2). For Fever, we set $\Gamma := 2(x+1)\Delta$. We note that this change in Γ will not impact protocol performance in the optimistic case that leaders are honest. We also observe below that this factor of 2 can also be decreased to arbitrarily close to 1 by making a simple change to the protocol.

The non-standard clock assumption. The assumption that Fever rests on is that, at the start of the protocol execution, $\mathtt{hg}_{f+1,0} \leq \Gamma$. The protocol also assumes that an honest processor's local clock progresses in real time unless 'bumped forward' (according to the protocol instructions). Given this assumption, the protocol is then designed so that, even though processors often bump their clocks forward:

(a) $\mathtt{hg}_{f+1,t} \leq \Gamma$ for all $t \geq 0$.
(b) If $\mathtt{hg}_{f+1,t} \leq \Gamma$ at $t \geq$ GST which is the first time an honest processor enters the initial view v with honest leader, then the leader will produce a QC.

Since the instructions are very simple, we just state them, and then show that they function as intended.

Initial and non-initial views. Fever does not consider any notion of epochs. The leader for view v is processor $\lfloor v/2 \rfloor \bmod n$. If v is even, then v is called 'initial', otherwise v is 'non-initial'. The reason we consider initial and non-initial views will become clear when we come to verify (b) above. As in Section 3.2, the clock-time corresponding to view v is $\mathtt{c}_v := \Gamma v$.

When processors enter views. If v is initial, then p enters view v when $\mathtt{lc}(p) = \mathtt{c}_v$. If v is not initial, then p enters view v if it is presently in a view $< v$ and it receives a QC (formed by the underlying protocol) for view $v-1$.

View Certificates. When an honest processor p enters a view v which is initial, it sends a $\mathtt{view}\ v$ message to $\mathtt{lead}(v)$. This message is just the value v signed by p. Once $\mathtt{lead}(v)$ receives $f+1$ $\mathtt{view}\ v$ messages from distinct processors, it combines these into a single threshold signature, which is a View Certificate (VC) for view v, and sends this VC to all processors.

When processors bump clocks. At any point in the execution, if an honest processor p receives a QC for view $v-1$ (formed by the underlying protocol) or a VC for view v, and if $\mathtt{lc}(p) < \mathtt{c}_v$, then p instantaneously bumps their local clock to $\mathtt{c}_v$.

Verifying the claim (a) above. Since the local clocks of honest processors only ever move forward, it follows that at any point in an execution, if an honest processor p has already contributed to a QC or a VC for view v, then $\mathtt{lc}(p) \geq \mathtt{c}_v$. To prove the claim, suppose towards a contradiction that there is a first point of the execution, t say, at which the claim fails to hold. Then it must be the case that some honest p bumps its clock forward at t, and that $p = p_{1,t}$ after bumping its clock forward, with $\mathtt{lc}(p)$ strictly greater than the value of any honest clock at any time $< t$.

There are two possibilities:

(1) p bumps its clock because it receives a VC for some view v with $\mathtt{c}_v > c(p)$. In this case, there must exist at least one honest processor $p' \neq p$ which contributed to the VC for view v. This contradicts the fact that $p = p_{1,t}$.
(2) p bumps its clock because it sees a QC for some view $v-1$. In this case, p bumps its clock to $\mathtt{c}_v$. At least $f+1$ honest processors must have contributed to the QC, which directly gives

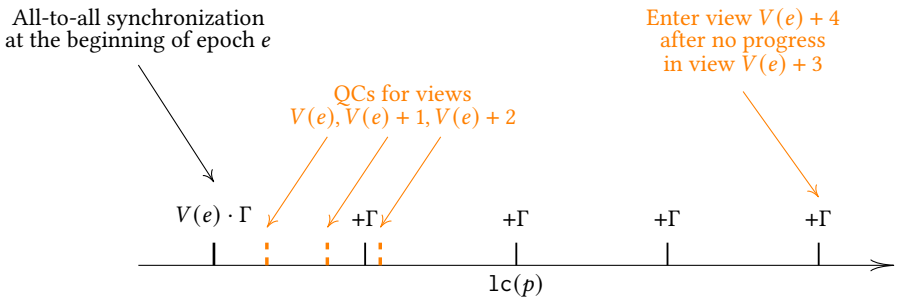

Figure 1: LP22: Epoch-synchronization and optimistically responsive QC generation

the required contradiction because each of those processors p' must have $\mathtt{lc}(p') \geq \mathtt{c}_{v-1} = \mathtt{c}_v - \Gamma$.

Verifying the claim (b) above. Suppose that $\mathtt{hg}_{f+1,t} \leq \Gamma$ at t which is the first time an honest processor enters an initial view v with honest leader. Then $\mathtt{lead}(v)$ will receive $f+1$ $\mathtt{view}\ v$ messages by time $t+\Gamma+\Delta$, and all honest processors will be in view v by time $t+\Gamma+2\Delta$ (if the leader has not already produced a QC for view v by this time). Note that processors do not enter view $v+1$ until seeing a QC for view v because view $v+1$ is not initial: this is key to ensuring all honest processors will be in view v (and not any higher view) by time $t+\Gamma+2\Delta$. All processors will then receive a QC for view v by time $t+2\Gamma-x\Delta$ and will be in view $v+1$ by this time (unless the leader has already produced a QC for view $v+1$). All processors will then receive a QC for view $v+1$ by time $t+2\Gamma$.

Although we do not give a formal proof here (we give a formal proof for Lumiere in the online version of the paper at [14]), it is also clear that the protocol is smoothly optimistically responsive because the delay caused by each faulty leader is at most Γ per view.

Reducing Γ. It is not difficult to see that Γ can be made arbitrarily close to $(x+1)\Delta$ by increasing the number of consecutive views allocated to each leader (and altering the definition of initial and non-initial views accordingly): doing so increases worst-case latency, but proportionally decreases the total time that can be wasted by faulty leaders.

3.4 Basic Lumiere Solution

To describe the full version of Lumiere, we break the presentation down into two steps. In this section, we show how to combine the techniques developed by LP22 and Fever to give a protocol called Basic Lumiere, which maintains $O(n^2)$ worst-case communication complexity while also being smoothly optimistically responsive. Then, in Section 3.5, we describe how to modify the protocol to remove the need for views with $\Omega(n^2)$ communication complexity within time $O(n\Delta)$ of GST (this last step turns out to be the one that is complicated).

The basic idea. The idea behind combining LP22 and Fever is simple. Fever requires the assumption that $\mathtt{hg}_{f+1,t} \leq \Gamma$ at the start of the protocol execution. While we do not wish to make this assumption, the 'heavy' synchronization process that LP22 employs at the start of each epoch *does* ensure $\mathtt{hg}_{f+1,t}$ is bounded by $\Delta < \Gamma$ at the start t of any epoch that begins after GST. We can therefore employ the Fever protocol *within epochs*. Doing so ensures that $\mathtt{hg}_{f+1,t'}$ remains bounded by Γ for $t' \geq t$ within each epoch. All honest leaders therefore produce QCs, and the protocol is smoothly optimistically responsive because each faulty leader can only cause Γ delay per view.

Initial, non-initial, and epoch views. The leader for view v is processor $\lfloor v/2 \rfloor \bmod n$. If v is even, then v is called initial: otherwise v is non-initial. If $v \bmod 2(f+1) = 0$, then v is called an epoch view. We set $V(e) := 2(f+1)e$. The clock-time corresponding to view v is $\mathtt{c}_v := \Gamma v$.

When processors enter initial non-epoch views. If v is initial and is not an epoch view, then p enters view v when $\mathtt{lc}(p) = \mathtt{c}_v$.

When processors enter non-initial views. If v is not initial, then p enters view v if it is presently in a view $< v$ and it receives a QC for view $v-1$.

When processors enter epoch views. Processor p enters the epoch view v if it is presently in a lower view and if p receives an EC for view v.

View Certificates. When an honest processor p enters a view v which is initial and which is not an epoch view, it sends a $\mathtt{view}\ v$ message to $\mathtt{lead}(v)$. This message is just the value v signed by p. Once $\mathtt{lead}(v)$ receives $f+1$ $\mathtt{view}\ v$ messages from distinct processors, it combines these into a single threshold signature, which is a view certificate (VC) for view v, and sends this VC to all processors.

Epoch certificates. Let $v := V(e)$. Processor p wishes to enter epoch e and view v if it is presently in a lower view and once its local clock reaches $\mathtt{c}_v$. At this point, it pauses its local clock and sends an epoch $\mathtt{view}\ v$ message to all processors. Upon receiving epoch $\mathtt{view}\ v$ messages from $2f+1$ distinct processors while in a view $< v$, any honest processor combines these into a single threshold signature, which is called an EC for view v, and sends the EC to all processors.

When processors bump clocks. At any point in the execution, if a correct processor p receives a QC for view $v-1$ (formed by the underlying protocol) or a VC or EC for view v, and if $\mathtt{lc}(p) < \mathtt{c}_v$, then p instantaneously bumps their local clock to $\mathtt{c}_v$. Upon receiving an EC for view v while in a lower view, p unpauses its local clock (if paused).

3.5 Removing Epoch Synchronisations for the Steady State

The idea behind removing the need for repeated 'heavy' ($\Theta(n^2)$ communication) epoch view changes after GST is that, once the first has been carried out, honest processors are already sufficiently synchronised. Since GST is unknown, however, processors cannot know directly when synchronization has occurred. Instead, they should look to see when some 'success criterion' has been satisfied. For example, one might wait to see an epoch which has produced a QC or a certain number of QCs, and then temporarily pause heavy view changes until one sees an epoch which does not satisfy the success criterion. Unfortunately, taking this approach immediately introduces some complexities, described below. We note that the following discussion considers an as yet unspecified 'success criterion': we will define an appropriate criterion once the relevant issues have been explained.

Some processors may see the success criterion satisfied, while others do not. In the case that the success criterion is satisfied due to QCs produced by faulty leaders, it may be the case that some honest processors fail to see the success criterion satisfied. Since they will then require an EC to enter the next epoch, those processors who do see the success criterion satisfied will still need to contribute to the EC. On the other hand, we do not wish Byzantine processors alone to be able to trigger EC formation, otherwise the Byzantine players will be able to cause every epoch to begin with a heavy view change. To deal with this, we have to modify the epoch change process slightly:

- If an honest processor sees the success condition satisfied for an epoch e, then they view $V(e+1)$ as a standard initial view (meaning that they enter the view when their local clock reaches $\mathsf{c}_{V(e+1)}$) and do not immediately send an epoch `view` $V(e+1)$ message (nor pause their local clock).
- Any honest processor who reaches the end of epoch e and does not see the success criterion satisfied pauses its local clock and sends an epoch `view` $V(e+1)$ message to all processors Any set of $f+1$ epoch `view` $V(e+1)$ messages from distinct processors is referred to as a TC for view v.
- When any honest processor in an epoch $\leq e+1$ sees a TC for view $V(e+1)$, they send an epoch `view` $V(e+1)$ message to all processors.
- Any processor that does not see the success criterion for epoch e satisfied enters epoch $e+1$ upon seeing an EC for view $V(e+1)$, and unpauses its local clock at that point.
- An EC for view $V(e+1)$ is now defined to be a set of epoch `view` $V(e+1)$ messages from $2f+1$ distinct processors.

The success criterion might be satisfied for all epochs after GST without synchronization actually occurring. Since QCs may be formed with the help of Byzantine processors, the formation of QCs does not actually imply that the $(f+1)^{\text{st}}$ honest gap is less than Γ (or even small). One is therefore potentially presented with a scenario where the success criterion continues to be satisfied for every epoch after GST with the help of Byzantine processors. If epochs involve $2(f+1)$ views, and if the success condition is seeing a single leader in the epoch produce two QCs, then this would represent highly sub-optimal behaviour. The situation can be improved by increasing the length of an epoch by a constant factor to $2n$ views (meaning each processor gets two successive views as leader) and by setting the success criterion to be $2f+1$ leaders each producing two QCs (for views in the epoch). If every epoch after GST produces the success criterion, this might now look like a reasonable outcome. Unfortunately, it still represents sub-optimal behaviour because f honest leaders may fail to produce QCs in each epoch if f Byzantine leaders each produce two QCs (in this case, the adversary is highly over-represented in QC generation).

To remedy this issue, the basic idea is to set Γ so that each honest leader who produces QCs is able to shrink the $(f+1)^{\text{st}}$ honest gap. If the success criterion continues to be satisfied, meaning that multiple honest leaders are shrinking the $(f+1)^{\text{st}}$ honest gap, then the aim is that, within expected time $O(n\Delta)$ of GST, the $(f+1)^{\text{st}}$ honest gap should come down below Γ.

Setting Γ to shrink the honest gap. For technical reasons we set $\Gamma := 2(x+2)\Delta$, but the following argument would also work for the value of Γ used in Section 3.3. We insist that honest leaders only produce a QC for view v if they can do it within time $\Gamma/2 - 2\Delta$ of sending the VC for view v, or within that time of sending the QC for the previous view if v is not initial. Note that this bound is on the time at which the QC is produced, rather than the time at which it is received.

To see that an honest leader who produces a QC after GST shrinks the $(f+1)^{\text{st}}$ honest gap, we can reason as follows: a more formal proof is given in the full proof of correctness in the online version of the paper at [14]. Let t be such that $\mathtt{lc}(p_{f+1,t}, t) = c_v$ and suppose $p := \mathtt{lead}(v)$ is honest. The instructions ensure that p receives $f+1$ `view` v messages by $t+\Delta$ and sends a VC to all processors by this time. The QC is then produced by time $t + \Gamma/2 - \Delta$ and is received by all processors by time $t + \Gamma/2$. Upon receiving this QC, honest processors whose local clocks are less than c_{v+1} forward their clocks to this value. This reduces the $(f+1)^{\text{st}}$ honest gap (and also the $(2f+1)^{\text{st}}$) by at least $\Gamma/2$ or to a value below Γ unless, for some $t' \in [t, t+\Gamma/2]$, some honest processor bumps its honest clock forward upon seeing a QC at t' and becomes $p_{1,t'}$ upon doing so. In the latter case the $(f+1)^{\text{st}}$ honest gap is anyway reduced to below Γ.

The honest gap must be brought below Γ within a single epoch. While the $(f+1)^{\text{st}}$ honest gap cannot be increased (except to a value below Γ) within an epoch, the same is not true of the $(2f+1)^{\text{st}}$ honest gap. Unfortunately, the fact that some honest processors may see epoch e produce the success criterion, while others do not, means that the process of moving to epoch $e+1$ may significantly increase the $(f+1)^{\text{st}}$ honest gap: If only f honest processors see the success criterion, then some honest processors will have to wait to see an EC for view $V(e+1)$ before entering the next epoch. This means that the $(f+1)^{\text{st}}$ honest gap can increase to become equal to the $(2f+1)^{\text{st}}$ honest gap.

If the first honest processor to enter epoch $e \geq 0$ does so after GST, then we will be able to show that the $(f+1)^{\text{st}}$ honest gap is less than $(4f+2)\Gamma$ at the start of the epoch. The difficulty described above can therefore be addressed by increasing the length of an epoch by a constant factor, so that a single epoch that produces the success criterion brings the $(f+1)^{\text{st}}$ honest gap to within Γ by the end of the epoch: since the $(f+1)^{\text{st}}$ honest gap is $<$

$(4f+2)\Gamma$ at the start of the epoch, decreasing it by $\Gamma/2$ at least $8f+2$ times suffices. We note that increasing the length of the epoch does represent a slight tradeoff. While Basic Lumiere outperforms LP22 on all measures, the cost for Lumiere of significantly decreasing the eventual worst-case communication complexity is a constant factor increase in the worst-case latency, i.e. the time to the *first* consensus decision after GST. Latency in the steady state is still significantly decreased when compared to LP22, because Lumiere is smoothly optimistically responsive.

Two further complexities. Extending the length of an epoch allows us to ensure that the $(f+1)^{\text{st}}$ honest gap is brought down to below Γ by the last view of the epoch. If the $(2f+1)^{\text{st}}$ honest gap is still large at that point, however, there remains the danger that the process of changing epoch will substantially increase the $(f+1)^{\text{st}}$ honest gap: We wish to ensure that the $(f+1)^{\text{st}}$ honest gap remains small thereafter, so that all honest leaders from QCs from that point on. To ensure that the epoch change does not increase the $(f+1)^{\text{st}}$ honest gap (or at least not by more than Δ), it suffices that the last leader of the epoch and the first leader of the next are both honest, since then they will be able to reduce the $(2f+1)^{\text{st}}$ honest gap to below Γ. To make things simple, we can also ensure that these two views have the same leader.

To describe a final complexity that needs to be dealt with, let us suppose that the $(f+1)^{\text{st}}$ honest gap is less than Γ at the beginning of an epoch e, so that all honest leaders produce QCs during the epoch. If time less than Δ passes after the success criterion is satisfied and before the clocks of some honest processors reach $\mathsf{c}_{V(e+1)}$ (because a long sequence of QCs are produced in time less than Δ), then it is possible that honest processors will not have seen the QCs produced by some honest leaders when their local clock reaches $\mathsf{c}_{V(e+1)}$, and will therefore send epoch $\mathtt{view}$ $V(e+1)$ messages. To remedy this obstacle, we have processors wait time Δ before sending an epoch $\mathtt{view}$ $V(e+1)$ message when they do not immediately see satisfaction of the success criterion.

4 THE FORMAL SPECIFICATION

It is convenient to assume that, whenever any processor sends a message to all processors, it also sends this message to itself (and this message is immediately received).

Local clocks. Recall that each processor p has a *local clock* value, denoted $\mathtt{lc}(p)$. Initially, p sets $\mathtt{lc}(p) := 0$. For simplicity, we suppose $\mathtt{lc}(p)$ advances in real time (with zero drift after GST), except when p pauses $\mathtt{lc}(p)$ or bumps it forward. As noted in Section 2, our analysis is easily modified to deal with a scenario where local clocks have bounded drift during any interval after GST in which they are not paused or bumped forward. Recall also Definition 3.1, which applies unaltered to this section.

Leaders and the clock time corresponding to each view. To determine the leader of view v, let $(g_0, \dots, g_{z-1})$ be a random ordering of the permutations of $\{1, \dots, n\}$ subject to the condition that, if $i \leq z$ is odd, then g_i and $g_{i+1 \bmod z}$ are reverse orderings.[2] We wish to give each leader two consecutive views, and to order leaders according to g_0, then g_1, and so on, cycling back to g_0 once we have ordered according to g_{z-1}. To achieve this, set $j := \lfloor v/(2n) \rfloor \bmod z$. The leader for view v, denoted $\mathtt{lead}(v)$, is processor $g_j(\lfloor v/2 \rfloor \bmod n)$.

Initial and non-initial views. If v is even, then view v is called *initial*. Otherwise, v is *non-initial*. The clock time corresponding to view v is $\mathsf{c}_v := \Gamma v$. We set $\Gamma := 2(x+2)\Delta$, and insist that honest leaders only produce a QC for view v if they can do it within time $\Gamma/2 - 2\Delta$ of sending the VC for view v, or within that time of sending the QC for the previous view if v is not initial. The operational distinction between initial and non-initial views is expanded on below.

Epochs and epoch views. Epoch e consists of the $10n$ views in the interval $[5(2n)e, 5(2n)(e+1))$. The first view of each epoch is called an *epoch view*, while other views are referred to as *non-epoch* views. We define $V(e) := 10ne$ and $E(v) := \lfloor v/(10n) \rfloor$, so that $V(e)$ is the first view of epoch e and $E(v)$ is the epoch to which view v belongs.

When processors enter initial views. A processor p enters the initial view v when $\mathtt{lc}(p) == \mathsf{c}_v$ if its local clock is not paused. Upon doing so, p sends a $\mathtt{view}$ v message to $\mathtt{lead}(v)$.

Forming VCs. Suppose v is an initial (epoch or non-epoch) view. If $\mathtt{lead}(v)$ is presently in view $v' \leq v$ and receives $\mathtt{view}$ v messages from $f+1$ distinct processors, then $\mathtt{lead}(v)$ forms a threshold signature which is a VC for view v and sends this to all processors.

The instructions upon receiving a VC. If p is presently in view $v' < v$ and receives a VC for initial view v, then p sets $\mathtt{lc}(p) := \mathsf{c}_v$.

When processors enter non-initial views. A processor p enters the non-initial view v if it is presently in a lower view and if it sees a QC for view $v-1$.

The instructions upon receiving a QC. If p is presently in view v and sees a QC for view $v' \geq v$, then it sets $\mathtt{lc}(p) := \mathsf{c}_{v'+1}$ if $\mathtt{lc}(p) < \mathsf{c}_{v'+1}$.

The operational distinction between initial and non-initial views is this. Upon their local clocks reaching c_v, where v is an initial non-epoch view, processors perform light view synchronization to bring others into view v. Namely, they enter view v and immediately send a view message. On the other hand, processors do not enter the non-initial view $v+1$ unless they obtain a QC for view v; view $v+1$ serves as sort of a "grace period" allowing the initial view v preceding it to produce a QC even after some local clocks have reached c_{v+1}. Note that, upon obtaining a QC for view v, processors bump their local clocks (if lower) to c_{v+1} and proceed to enter view $v+1$ immediately if in a lower view. If a processor receives a QC for the non-initial view $v+1$ while in a view $\leq v+1$, then it immediately enters view $v+2$ (unless, perhaps, $v+2$ is an epoch view – see below).

The success criterion. Each processor p maintains a local variable $\mathtt{success}(e)$, initially 0. Processor p sets $\mathtt{success}(e) := 1$ upon seeing at least $2f+1$ distinct processors each produce 10 QCs for views in the epoch. We say that epoch e 'produces the success criterion' if at least $2f+1$ distinct processors each produce 10 QCs for views in the epoch.

[2] The latter condition on reverse orderings is stipulated so that, for $e \geq 0$, the last leader of epoch e is the same as the first leader of epoch $e+1$.

Algorithm 1 The instructions for processor p

Local variables
$\texttt{lc}(p)$, initially 0 ▷ This is the value of p's local-clock
$view(p)$, initially -1 ▷ The present view of p.
$epoch(p)$, initially -1 ▷ The present epoch of p.
$\texttt{success}(e)$, $e \in \mathbb{Z}_{\geq -1}$, initially 0 ▷ Updated as described above

▷ ——— Epoch Synchronization Instructions (start) ———

Upon first seeing $\texttt{lc}(p) == \texttt{c}_v$ for epoch view $v > view(p)$ and $\texttt{success}(E(v) - 1) == 0$:
    Pause local-clock $\texttt{lc}(p)$ until seeing an EC, QC or VC for a view $\geq v$ or a TC for a view $> v$, or until $\texttt{success}(E(v) - 1) == 1$;
    If local clock is still paused time Δ after pausing, send an epoch `view` v message to all processors;

Upon first seeing $\texttt{lc}(p) == \texttt{c}_v$ for epoch view $v > view(p)$ and $\texttt{success}(E(v) - 1) == 1$:
    Set $epoch(p) := E(v)$ and $view(p) := v$;

Upon first seeing a TC for epoch view v with $E(v) \geq epoch(p)$:
    If $\texttt{lc}(p) < \texttt{c}_v$ **then**:
        For each initial view v' with $view(p) \leq v' < v$ send a `view` v' message to $\texttt{lead}(v')$ if not already sent;
        Set $\texttt{lc}(p) := \texttt{c}_v$;
    If $view(p) < v - 1$ **then** set $view(p) := v - 1$ and $epoch(p) := E(v) - 1$;
    Send an epoch `view` v message to all processors if not already sent;

Upon first seeing an EC for epoch view v with $E(v) > epoch(p)$:
    Set $view(p) := v$ and $epoch(p) := E(v)$;

▷ ——— Epoch Synchronization Instructions (end) ———
▷ ——— View Synchronization Instructions (start) ———

Upon $\texttt{lc}(p) == \texttt{c}_v$ for v initial and $epoch(p) == E(v)$:
    If $view(p) < v$, set $view(p) := v$;
    Send a `view` v message to $\texttt{lead}(v)$;

If $p == \texttt{lead}(v)$ for initial view $v \geq view(p)$:
    Upon first seeing `view` v messages from $f + 1$ distinct processors:
        Form a VC for view v and send to all processors;

Upon first seeing a VC for initial view $v > view(p)$;
    If $\texttt{lc}(p) < \texttt{c}_v$ **then**:
        For each initial view v' with $view(p) \leq v' < v$ send a `view` v' message to $\texttt{lead}(v')$ if not already sent;
        Set $\texttt{lc}(p) := \texttt{c}_v$;
    Set $view(p) := v$, $epoch(p) := E(v)$;

▷ ——— View Synchronization Instructions (end) ———
▷ ——— bump local-clock forward on QC ———

Upon first seeing a QC for view $v \geq view(p)$:
    If $\texttt{lc}(p) < \texttt{c}_{v+1}$ **then**:
        For each initial view v' with $view(p) \leq v' < v$ send a `view` v' message to $\texttt{lead}(v')$ if not already sent;
        Set $\texttt{lc}(p) := \texttt{c}_{v+1}$.
    If $v + 1$ is a non-epoch view **then** set $view(p) := v + 1$ and $epoch(p) := E(v + 1)$;
    If $v + 1$ is an epoch view and $view(v) < v$ **then** set $view(p) := v$ and $epoch(p) := E(v)$;

ECs and TCs. Suppose $v = V(e)$. An EC for view v is a set of $2f + 1$ epoch `view` v messages, signed by distinct processors. A TC for view v is a set of $f + 1$ epoch `view` v messages, signed by distinct processors.

The instructions for entering epoch views. Suppose $v = V(e)$. When $\texttt{lc}(p) = \texttt{c}_v$, there are two cases:

- Upon first seeing $\texttt{lc}(p) = \texttt{c}_v$ and $\texttt{success}(E(v) - 1) = 0$:
  - p pauses its local-clock until seeing an EC, QC or VC for a view $\geq v$, or a TC for a view $> v$, or until $\texttt{success}(E(v) - 1) = 1$;
  - If its local clock is still paused time Δ after pausing, then p sends an epoch `view` v message to all processors.
- Upon first seeing $\texttt{lc}(p) = \texttt{c}_v$ and $\texttt{success}(E(v) - 1) = 1$ (and also under other conditions already stated in the bullet point above), p enters epoch e and view $V(e)$.

Forming ECs. Suppose $v = V(e)$. As stipulated above, p sends an epoch `view` v message to all processors Δ time after its local clock reaches $\texttt{lc}(p) = \texttt{c}_v$ if $\texttt{success}(e) = 0$ at this time. We further stipulate that if p is in epoch $e' \leq e$ and receives a TC for view v, then p sets $\texttt{lc}(p) = \texttt{c}_v$ if $\texttt{lc}(p) < \texttt{c}_v$ and sends an epoch `view` v message to all processors.

The pseudocode is shown in Algorithm 1.

The proof of correctness can be found in the online version of the paper at [14].

5 RELATED WORK

HotStuff [20] was the first BFT SMR protocol to separate the view synchronization module and the core consensus logic. HotStuff named the view synchronization module the "PaceMaker" and left its implementation unspecified. While HotStuff requires 3 round trips within each view, HotStuff-2 [15] reduces this number to 2 round trips.

Cogsworth [16] was the first to formalize BVS as a separate problem, and provided an algorithm with expected $O(n)$ communication complexity and expected $O(1)$ latency in runs with benign failures, but with sub-optimal performance in the worst case. Naor and Keidar [17] improved Cogsworth to runs with Byzantine faults and, when combined with Hotstuff, produced the first BFT SMR protocol with expected linear message complexity in partial synchrony. Both protocols suffer from sub-optimal cubic complexity in the worst case.

Several papers by Bravo, Chockler, and Gotsman [2, 4, 5] define a framework for analyzing the liveness properties of SMR protocols.

These papers do not attempt to optimize performance, but rather introduce a general framework in the partial synchrony model to allow better comparison of protocols. For example, they describe PBFT [6] within this framework.

Two recent protocols, RareSync [7] and LP22 [12], both solve BVS with optimal $O(n^2)$ communication complexity in the worst case (providing cryptographic assumptions hold), thereby finally matching the lower bound established by Dolev and Reischuk in 1985 [9]. LP22 also achieves optimistic responsiveness. However, both of these protocols suffer from two major issues. First, neither protocol is smoothly optimistically responsive. RareSync is not optimistically responsive. While LP22 is optimistically responsive, even a single Byzantine processor may infinitely often cause $\Omega(n\Delta)$ latency between consecutive consensus decisions. Second, even in the absence of Byzantine action, infinitely many views require honest processors to send $\Omega(n^2)$ messages. Albeit this communication overhead being amortized across $O(n)$ decisions, it may cause periodic slowdowns. Ideally, one would hope for worst-case complexity between every pair of consensus decisions which is (after some finite time after GST) $O(f_a n + n)$. For LP22 and RareSync, the fact that infinitely many views require honest processors to send $\Omega(n^2)$ messages means that the corresponding bound is $O(n^2)$.

Fever [13] is another recent protocol, which operates in a different model than RareSync and LP22. While Fever makes standard assumptions regarding message delivery in the partial synchrony model, the protocol requires stronger than standard assumptions on *clock synchronization*. Specifically, Fever assumes that there is a known bound on the gap between the clocks of honest processors at the start of the protocol execution, and that the clocks of honest processors suffer bounded drift prior to GST. Under these stronger and non-standard assumptions, Fever achieves optimal $O(n^2)$ communication complexity in the worst case and also addresses the two issues raised above. In comparison, Lumeiere achieves these results under standard clock assumptions.

6 FINAL COMMENTS

Lumiere introduces two fundamental innovations. The first of these combines techniques from [13] and [7, 12] to give a protocol with $O(n^2)$ worst-case communication complexity and which has eventual worst-case latency $O(f_a\Delta + \delta)$.

The second innovation removes the need for repeated heavy epoch changes, and results in a protocol with eventual worst-case communication complexity $O(nf_a + n)$. Since implementing this second change requires a constant factor increase in epoch length, it is most practically useful in contexts where periods of asynchrony are expected to be occasional and where synchrony reflects the standard network state.

We leave it as an open question as to whether it is possible to achieve a protocol with the same worst case communication complexity, eventual worst-case communication complexity and eventual worst-case latency as Lumiere, but which also achieves better than $O(n\Delta)$ worst-case latency.

REFERENCES

[1] 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.

[2] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2020. Making Byzantine Consensus Live. In *34th International Symposium on Distributed Computing*.

[3] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2022. Liveness and Latency of Byzantine State-Machine Replication. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 12:1–12:19. https://doi.org/10.4230/LIPIcs.DISC.2022.12

[4] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2022. Liveness and latency of Byzantine state-machine replication. In *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[5] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2022. Making byzantine consensus live. *Distributed Computing* 35, 6 (2022), 503–532.

[6] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.

[7] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 2022. Byzantine Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony!. In *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA (LIPIcs, Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:21. https://doi.org/10.4230/LIPIcs.DISC.2022.14

[8] Shir Cohen, Idit Keidar, and Oded Naor. 2021. Byzantine agreement with less communication: Recent advances. *ACM SIGACT News* 52, 1 (2021), 71–80.

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

[10] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (apr 1988), 288–323. https://doi.org/10.1145/42282.42283

[11] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[12] Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. *CoRR* abs/2201.01107 (2022). arXiv:2201.01107 https://arxiv.org/abs/2201.01107

[13] Andrew Lewis-Pye and Ittai Abraham. 2023. Fever: Optimal Responsive View Synchronisation. *arXiv preprint arXiv:2301.09881* (2023).

[14] Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. 2024. Lumiere: Making Optimal BFT for Partial Synchrony Practical. arXiv:2311.08091 [cs.DC] Full version of this paper.

[15] Dahlia Malkhi and Kartik Nayak. 2023. HotStuff-2: Optimal Two-Phase Responsive BFT. *Cryptology ePrint Archive* (2023).

[16] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2021. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems* 1, 2 (oct 22 2021). https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization.

[17] Oded Naor and Idit Keidar. 2020. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference (LIPIcs, Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:17. https://doi.org/10.4230/LIPIcs.DISC.2020.26

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

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

[20] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.