Invited Paper: Lessons from HotStuff

Dahlia Malkhi
Chainlink Labs

Maofan Yin
Chainlink Labs
Ava Labs, Inc.

ABSTRACT

This article will take you on a journey to the core of blockchains, their Byzantine consensus engine, where HotStuff emerged as a new algorithmic foundation for the classical Byzantine generals consensus problem. The first part of the article underscores the theoretical advances HotStuff enabled, including several models in which HotStuff-based solutions closed problems which were opened for decades. The second part focuses on HotStuff performance in real life setting, where its simplicity drove adoption of HotStuff as the golden standard for blockchain design, and many variants and improvements built on top of it. Both parts of this document are meant to describe lessons drawn from HotStuff as well as dispel certain myths.

CCS CONCEPTS

• **Software and its engineering → Software fault tolerance**; • **Security and privacy → Distributed systems security**.

KEYWORDS

Byzantine fault tolerance; consensus; blockchain; HotStuff

ACM Reference Format:
Dahlia Malkhi and Maofan Yin. 2023. Invited Paper: Lessons from HotStuff. In *The 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED 2023), June 19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3584684.3597268

1 INTRODUCTION

Every time you use a cloud service, there are servers behind the scenes keeping redundant copies of your data, solving the distributed *consensus* problem to keep information available and consistent against the breakdown of some servers; Every time you fly a modern airplane, there are extra sensors and avionics to keep it airborne, reaching a consensus on automated inputs to flight controls against malfunctioning components; At the core of blockchains are systems that also solve consensus, to collectively maintain an immutable history of transactions against the worst type of failures, "Byzantine", orchestrated by rogue participants.

For four decades, experts in the field of distributed computing searched for optimal solutions to the classical Byzantine Fault Tolerant (BFT) consensus problem [23]. Recently, a master thesis titled "Consensus in the Age of Blockchains" [6], which was looking for a blockchain solution that developers can understand, changed the way we think about the problem. It led to the introduction of HotStuff [43], the first practical solution (the meaning of "practical" is defined below) with optimal communication complexity, that emerged as a new algorithmic foundation for the classical BFT consensus problem and a golden standard for blockchains.

This article will take you on a journey from the emergence of HotStuff to lessons from it along two dimensions, foundational and applied. The first part, Sections 3–Section 5, underscores the theoretical advances HotStuff enabled, including several models in which HotStuff-based solutions closed problems which were open for decades. This part finishes off with a surprising recent observation, HotStuff-2 [28], demonstrating that it is possible to improve the original HotStuff latency by as much as 33% without sacrificing any of its desirable properties (Section 5). The second part, Section 6, focuses on HotStuff performance in real life settings, where its simplicity drove the adoption of HotStuff as the golden standard for blockchain design, and many variants and improvements built on top of it.

2 PRELIMINARIES

The Problem. Briefly, in log replication, a group of hosts referred to as nodes reach agreement on a growing sequence of bundled values called "blocks". For our purposes, a solution is viewed as "practical" if it maintains consistency against any unforeseen network delays and advances at network speed, namely, as soon as a certain threshold of messages are received from participants. This setting is known as *partially-synchronous*.

More specifically, partially-synchronous BFT consensus replicates a log among $n = 3f + 1$ nodes, f of which are Byzantine. Byzantine nodes may collude and deviate from the specified protocol arbitrarily, though still with some common constraints (e.g., cannot have infinite computational power). There is a known bound Δ on message transmission delays (neglecting message processing as marginal), such that after an unknown Global Stabilization Time (GST), all transmissions arrive within Δ bound to their destinations.

Nodes output increasing log prefixes with the following guarantees:

Safety At all times, for every pair of correct nodes, the output log of one is a prefix of the other.

Liveness After GST, all non-faulty nodes repeatedly output (growing) logs.

We additionally desire to simultaneously achieve $O(n^2)$ worst-case communication, optimistically linear communication, an optimistically fast latency, and optimistic responsiveness. We define these properties more formally below.

Performance measures. Theoretical complexity measures are evaluated after GST, since no progress is guaranteed until then. There are two principal complexity measures: communication, measured

ApPLIED 2023, June 19, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0128-3/23/06.
https://doi.org/10.1145/3584684.3597268

in the number of bits sent over communication channels (by one node or in total); and latency, measured in units of network delays, maximum (Δ) or actual (δ). We are interested in several aspects of these measures (communication and/or latency): expected, optimistic, and worst-case.

Measures expressed as expectations are taken over protocol coin tosses, notably for electing "leaders" internally (see Section 3). Optimistic performance measures are taken in faultless, synchronous executions. These measures also reflect the protocol performance after a certain stabilization time following GST, but this analysis is left out of this short paper. Worst-case performance measures are taken against an unlucky cascade of $O(n)$ (leader) failures.

The desirable performance goals, which are derived from several known lower bounds, are as follows:

Latency. A solution has *optimistic responsiveness* if optimistic latency is $O(\delta)$ per decision. An $\Omega(n\Delta)$ worst-case latency is mandated by the Aguilera-Toueg bound [4].

Communication. A solution is optimal in worst-case communication if it incurs $O(n^2)$ communication cost [13]. The best communication cost to optimistically reach is $O(n)$ (the lower bound is trivial).

Load-Balance. A solution has *load balance* if the same communication cost is incurred per party over a sequence of consensus decisions. Notably, this implies rotating leaders regularly.

It's worth noting that throughput is not a theoretical complexity measure. We discuss the throughput of various systems in Section 6.

3 WHY HOTSTUFF?

In order to understand the improvement HotStuff introduces, let us consider a brief evolution of practical BFT solutions that led to it and the scaling properties they targeted.

View-by-View Recipe. PBFT [9], a landmark in BFT solutions introduced two decades ago, emphasizes optimistically low latency. It established a view-by-view "recipe" that works as follows. A view consists of two abstract steps. In the first step, a designated *leader* attempts to *reconcile* an output value, and in the second step, nodes *ratify* if there is an agreement and commit it. An advantage of this leader-based regime is that it is optimistically responsive (defined in Section 2), that is, under synchronous faultless settings, it does not need to wait for the maximum Δ delay, it instead incurs the actual network delay δ. Therefore, PBFT exhibits a desirable feature, responsiveness, in optimistic settings:

F-1 Optimistic responsiveness

Linear Secure Broadcast. PBFT employs a *secure broadcast* building block to disseminate a leader proposal. A secure broadcast provides a guarantee that non-faulty nodes deliver the same message from a sender, if any, and that messages from non-faulty leaders are reliably delivered. A second secure broadcast is used for assembling $2f + 1$ votes to commit the proposal.

PBFT's original secure broadcast protocol is based on a protocol by Bracha [5] and has quadratic communication complexity. Two pioneering works in the field, VBA [8] and Rampart [36], which were later adopted in SBFT [17], employ signature aggregation for secure broadcast whose communication complexity is linear: A sender collects signatures on its message by a quorum of $2f + 1$ out of $n = 3f + 1$, aggregates the signatures into a Quorum Certificate (QC) and disseminates the QC.

Replacing PBFT's secure broadcast with a linear variant yields a two-step protocol depicted in Figure 1(left), each step a linear secure broadcast, and achieves the following feature:

F-2 Optimistic communication linearity

View-Change with Quadratic Complexity. If a leader fails or the network stalls (before GST) during the ratify step, as depicted in Figure 1(right), a new leader needs to check if any value is locked by a node from a previous view, and ratify it.

The ratify step in all the above protocols uses a lock-commit paradigm (aka commit-adopt [14]), where sufficiently many nodes are locked before any node can commit. If a new leader does not learn of any locked value, it can make a different proposal. However, if it turns out that some nodes are locked on another value, they nevertheless need to vote for the (safe) new proposal to allow progress. Consequently, in PBFT, a new leader must *prove* that $2f + 1$ nodes did not vote to commit a different proposal. This approach for justifying a new leader proposal after a view-change is the foundation of all protocols in the PBFT family, including FaB [29], Zyzzyva [21], Aardvark [11], SBFT [17], and most former protocols in the two-phase HotStuff family [3, 15, 16, 18, 40] except HotStuff-2, which we will get to later. Unfortunately, this justification proof is complex to code and incurs quadratic communication complexity.

Simplified View-Change without Responsiveness. Tendermint [6] introduced a simpler view-change sub-protocol than PBFT, later adopted in Casper [7]. A new leader proposal simply hinges on the latest locked value (the highest block receiving a QC) the leader knows. In fact, this simplification turns a new leader sub-protocol identical to a steady leader sub-protocol. That is, in Tendermint there is no explicit view-change sub-protocol. This provides another crucial tenet for blockchains: rotating leaders routinely, balancing participation and control among all nodes, as captured by the following feature:

F-3 Balanced communication load over sequences of decisions

However, to guarantee that a leader obtains the latest locked value in the system, a leader in Tendermint has to wait for the maximum network delay Δ. Hence, it does not satisfy optimistic responsiveness (F-1), namely, each view sub-protocol incurs an explicit delay for the maximum network latency. Moreover, the view sub-protocol is simpler but has the same complexity as PBFT, $O(n^2)$. Nevertheless, Tendermint provided a crucial step in simplifying the view-change that is harnessed in HotStuff.

Linear, Simple View-Change with Responsiveness. HotStuff [43] harnesses and enhances the simple Tendermint view-change in the following manner:

First, it removes the need for each view-change to delay, thereby satisfying F-1 in addition to F-2 and F-3. This is achieved by employing three consecutive secure broadcasts, instead of two, to form a decision, as depicted in Figure 2. The first broadcast forms a QC guaranteeing the uniqueness of a leader proposal; the second provides $2f + 1$ nodes with a copy of the QC (referred to as "key") to

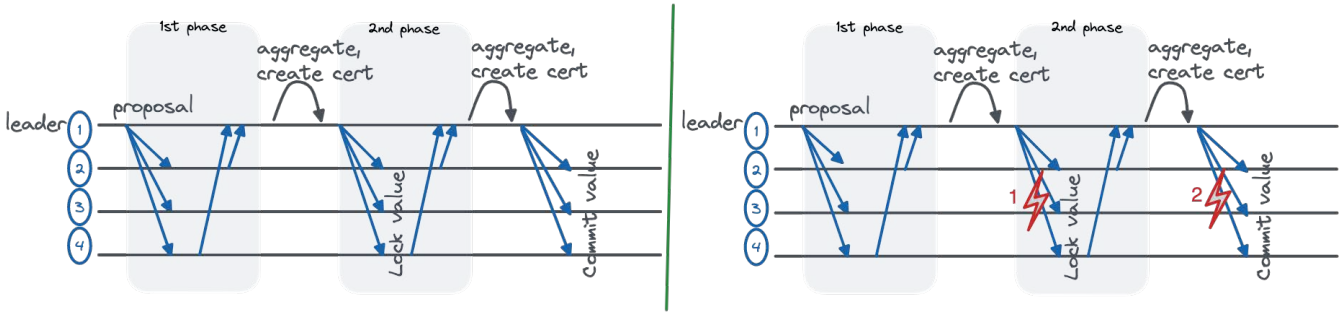

Figure 1: Two-step protocol, each step a linear secure broadcast (left). Possible failures during ratify step (right).

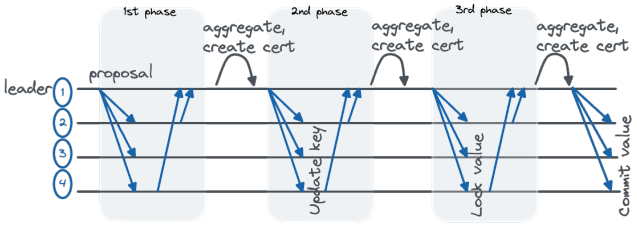

Figure 2: HotStuff three-step protocol.

pass to the next leader, before any node can become locked or commit a value; the third confirms that $2f + 1$ have a key and commits the value.

In a way, HotStuff spreads the lock-commit ratification step over two linear secure broadcasts. The extra phase guarantees that if any party is locked on a leader proposal, then $2f + 1$ already obtained a key corresponding to this lock. Correspondingly, the next leader would learn about the latest lock even if f are Byzantine. In Figure 3, the new leader (party 2) obtains the key from party 3 (Byzantine party 4 may not send its key), despite party 3 itself not reaching the lock stage.

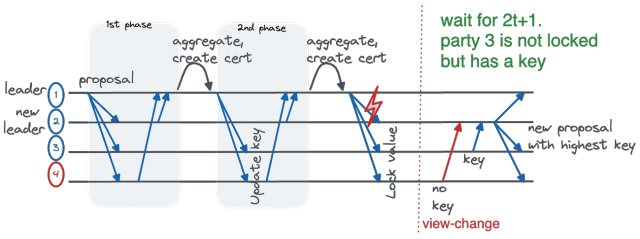

Figure 3: HotStuff view-change scenario.

Second, HotStuff employs linear secure broadcasts to spread a leader proposal, making the view-change linear.

Third, a view-change sub-protocol must additionally address view-synchronization, referred to as a *Pacemaker* in [43]. A Pacemaker coordinates for nodes to enter the next view roughly at the same time as the leader in order to guarantee progress. RareSync [10] and Lewis-Pye [25] demonstrate a Pacemaker, which was mentioned only at a high-level in HotStuff [43], that has worst-case $O(n^2)$ communication complexity.

Jointly, these enhancements achieve the following feature:

F-4 Worst-case communication optimality

In summary, all the mentioned desirable performance properties (F-1,2,3,4) are simultaneously achieved by HotStuff with an optimal Pacemaker. It is worth noting that the HotStuff family of protocols suffers an extra phase within the view sub-protocol compared with PBFT and Tendermint. We will come back to this in Section 5.

4 HOTSTUFF KEY CONTRIBUTIONS

4.1 Pipelining

An important property stemming from the simplified leader replacement protocol is that all three secure broadcast steps of HotStuff are essentially identical. This led to a key contribution introduced in HotStuff, namely, pipelining the protocol over a chain of blocks, each block embodying one step of the protocol. Furthermore, each block can be proposed by a different leader.

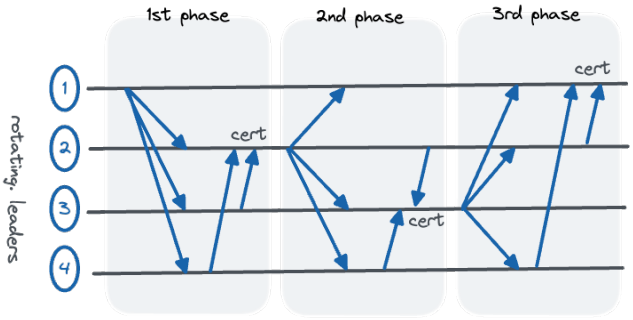

Figure 4: Pipelining.

Each block in a pipeline is constructed by a leader proposal in one view and becomes certified via secure broadcast. The next view proposes a block which is chained to the previous one, constituting the second step of the first proposal, and simultaneously, the first step of a new proposal. And so on. Figure 4 depicts a pipeline of three blocks, the first of which becomes committed.

The most important outcome of HotStuff pipelining is that it is easy to understand how the protocol constructs a replicated chain of blocks. Figure 5 below provides an easy visual explanation of

the HotStuff three-chain rule: whenever the depicted three-block pattern occurs, the head of the three-chain becomes committed.

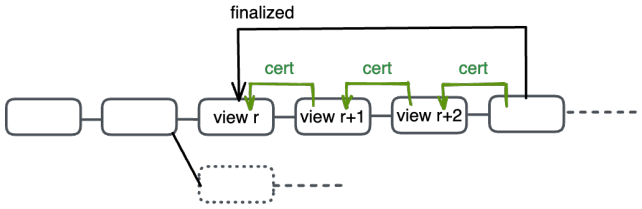

Figure 5: Three-chain commit rule.

Prior to HotStuff, log replication solutions reached consensus one position at a time via a multi-round protocol, in a notoriously sophisticated fashion [1, 30]. Contrarily, HotStuff is apparent and intuitive, one simply looks for an uninterrupted chain of blocks to identify a consensus decision. View changes that are necessary to resume the replication are depicted by forks from the main branch.

HotStuff also manages to encode almost all of its protocol state into the data (i.e., blocks) it replicates, reducing to just two types of messages: block proposals and votes. The execution of the protocol and final commitment are made solely by checking the immutable chain fabrication that implicitly represents a consistent causal ordering among all messages. This inspired "zero-cost" consensus protocols [12, 37] that also use blocks to vote and thus entirely operate upon the data it replicates without extra message exchange.

4.2 Linearity

The linear fast path and view-change subprotocols of HotStuff empowered several tight solutions to open challenges in the consensus arena.

Partially-synchronous BA. The most direct tight results enabled by HotStuff are RareSync [10] and Lewis-Pye [25], the first optimal partially-synchronous Byzantine agreement solutions, whose worst-case communication complexity is $O(n^2)$ with $O(n\Delta)$ latency (recall, worst-case complexities are taken after GST, against a cascade of f actual leader failures). Both solutions address HotStuff's view-synchronization black-box component, solving it with both expected and worst-case $O(n^2)$ communication complexity.

Asynchronous BA. VABA [2] is the first optimal solution to the long standing validated asynchronous Byzantine agreement problem[1] whose communication complexity is $O(n^2)$.

VABA invokes n simultaneous HotStuff consensus instances, where each node acts as the leader. After $n - f$ instances complete, VABA elects in retrospect one instance unpredictably and uniformly at random. It either has reached a decision by its leader, or orchestrates n view-changes from it to the next wave of n instances. Running n simultaneous views and electing a random leader in retrospect was suggested before in [20], but the HotStuff linear view-change enabled managing n view-changes with overall complexity $O(n) \cdot n = O(n^2)$.

[1]In a nutshell, validated agreement enforces an external validity predicate on decisions, rather than the theoretical Byzantine agreement problem formulation requiring all nodes to start with the same input.

Optimistically Asynchronous BA. Bolt-Dumbo [26] and Jolteon and Ditto[15] demonstrate an optimistically asynchronous Byzantine agreement, a problem pioneered in [22, 35]. They use a two-phase variant of HotStuff as an optimistically linear path, for the case of a non-faulty leader and partial synchrony settings. They employ a quadratic asynchronous protocol as the fallback upon a leader failure, thereby providing resilience against asynchrony.

4.3 The Pacemaker Module

The *Pacemaker* abstraction introduced by HotStuff captures the view synchronization challenge as a separate module in BFT consensus. This modularity contributed further to HotStuff developer-friendliness. Additionally, the formulation of the Pacemaker as a problem in itself has sparked interest, leading to several advances.

Briefly, a Pacemaker solves the Byzantine view synchronization problem, where a group of processes enters/leaves views until it reaches a view with a non-faulty leader and spends sufficient overlapping time in the view for the leader to drive a consensus decision. Before HotStuff, BFT solutions for the partial synchrony settings required quadratic communication complexity per view-change, hence no one cared if coordinating view advancement also incurs quadratic communication. Linearity has shifted the challenge to developing a Pacemaker with low communication.

Cogsworth [32] and a protocol by Naor and Keidar (NK) [33] demonstrate Pacemakers with expected linear communication complexity whose worst-case is $O(n^3)$. Expected linearity is achieved via the following strategy. When nodes want to move to the next view, they send a message only to the next view's leader. The leader collects the messages from the nodes, and once it receives enough messages, it combines them into a threshold signature and sends it to the nodes. This all-to-leader, leader-to-all communication pattern is similar to the one used in HotStuff; the trick in Cogsworth/NK is utilizing $f + 1$ consecutive leaders as fallback *relayers*, staggering leaders one at a time—each after a (tunable) Pacemaker timeout, until there is progress. One of the relayers is non-faulty and will facilitate entering the next view.

Two recent works, RareSync[10] and Lewis-Pye (LP) [25], solve the view synchronization problem with both expected and worst-case $O(n^2)$ communication complexity. Both use a similar approach, which is remarkably simple and elegant. It bundles consecutive views into epochs, where each epoch consists of $f + 1$ consecutive views. Nodes employ a Bracha-like all-to-all coordination protocol in the first view of each epoch, and then they advance through the rest of the views in the same epoch using timeouts if there is no progress in the underlying consensus protocol. The downside of RareSync/LP is that the expected message complexity and latency are as bad as the worst case, hence the expected case performance is worse than previous solutions.

It remains open and an active area of research to find view-synchronization solutions with both optimal worst-case and expected/optimistic performance. Further discussion of view synchronization appears in [27].

5 TWO-PHASE HOTSTUFF

Since the introduction of HotStuff it remained an open challenge to achieve the desirable properties F-1,2,3,4 it encompasses with a

two-phase view rather than a three-phase sub-protocol. Recently, a two-phase HotStuff variant named HotStuff-2 was introduced in [28] showing it is possible to simultaneously achieve all five desirable properties. That is, it is possible to solve partially-synchronous BFT and simultaneously achieve a two-phase commit sub-protocol within a view, optimistic responsiveness, optimistic communication linearity, balanced load across nodes, and $O(n^2)$ worst-case communication. The main takeaway is that two phases are enough for BFT after all.

HotStuff-2 is remarkably simple, adding no substantial complexity to the original HotStuff protocol. It builds on two secure broadcasts. The first step certifies with a QC uniqueness of a leader proposal. The second one is a lock-commit step for ratifying it.

The key observation is that a new leader can choose between two options: If the leader obtains a QC from the preceding view, it **knows** that it has obtained the latest locked value that possibly exists in the system. In this case, it proceeds with a proposal in a responsive manner. Otherwise, the leader **knows** that a timer delay of Δ must have expired in the preceding view. In that case, there is no responsiveness anyway, hence it waits an extra Δ to obtain the latest locked value in the system. Figure 6 depicts two possible HotStuff-2 view-change scenarios.

Prior to HotStuff-2, there has been a long line of HotStuff variants aiming to improve HotStuff's view regime to two phases. Fast HotStuff [18], DiemBFT-v4 [40], and Jolteon and Ditto [15], provide two-phase view regimes but revert to a PBFT quadratic view-change (Ditto also adds resilience against asynchrony, as mentioned above). Hence, they do not satisfy F-4, namely they incur $O(n^2)$ communication every time a leader is faulty. A fortiori, an unlucky cascade of faulty leaders incurs $O(n^3)$ communication. Wendy [16] and MSCFCL [3] also revert to a PBFT view-change with a leader proof to convince parties of a safe proposal, but focus on compressing the leader proof. These schemes employ somewhat heavy hammers: Wendy introduces a novel signature scheme that works only when the gap between views that make progress is constant-bounded and MSCFCL utilizes succinct arguments of knowledge whose complexity blows up quickly. All of these advances are much more complex than HotStuff-2, whose suprising upshot is that none of them is necessary.

6 SCALING

Aside from theoretical considerations, practical consensus protocols also need to be fast and scalable when it comes to actual implementation. Over the past years, as we learned about HotStuff variants and studied subsequent protocols, we extracted several insights about improving the performance of HotStuff and discovered some prevailing myths about scalability.

The main scalability challenge is the overhead of coordination among an increasing number of participating nodes and increasing network latencies among them. The goal is to maintain high throughput and low latency.

6.1 What is the "Leader Bottleneck"

The principal reason for using leader-based consensus protocols in general, not just HotStuff, which we've heard repeatedly from multiple blockchain projects, is the emphasis on low latency. In particular, using the reconcile/ratify consensus recipe described in Section 5, a good leader can drive the reconciliation step in one network round-trip, and in just one more (logical) step, agreement can be detected and committed. However, one of the strongest weaknesses mentioned in the literature is the so-called "leader bottleneck".

Specifically in HotStuff, the leader bottleneck is manifested in a pipeline of linear secure broadcasts. In each instance in the pipeline, first, a leader disseminates blocks and all other nodes are not communicating with one another, thereby the network bandwidth is underutilized; second, the leader collects signed messages from all nodes, validates (aggregates) the signatures, and updates its protocol state, while all other nodes are idle. Linear secure broadcasts are invoked in a sequence, where each one has to wait for responses from $2f + 1$ nodes before it moves to the next step. This takes a full round-trip to and from the slowest node among the fastest $2/3$ of the network. In WAN settings with geo-distributed nodes, this almost always takes an order of hundreds of milliseconds, including additional time spent verifying $2f + 1$ votes.

At first glance, it thus appears that low latency comes at the cost of bounded throughput.

We proceed to describe prevailing approaches for parallelizing work in order to saturate network and computational resources. Some approaches are compatible with HotStuff and may be harnessed to increase its throughput; others hinge on new BFT consensus foundations.

6.2 Saturating the Resources

Parallel Computation. A simple way to increase throughput is to offload networking and computationally intensive tasks to *workers*. Despite the sequential skeleton of a consensus protocol, signature verification, "mempool" (a blockchain subsystem which buffers transactions from clients and bundles them into blocks) synchronization, and/or block dissemination, can be made parallel in between the key phases of the consensus. For example, we heard that from many real-life HotStuff systems that the leader work is offloaded to a farm of CPUs or even to a local cluster of hosts, each handling messages to/from other nodes and carrying verification in parallel.

Large Blocks. Another simple way to increase throughput and ameliorate the idle time caused by network latency is to batch larger payloads per block. The key insight here is that the non-network time required to handle/process/execute a block grows linearly with block size, whereas network transmission time remains almost fixed, or grows very slowly. This means that the utilization rate increases by larger blocks and throughput grows. However, although this will increase throughput it will also increase latency. Additionally, larger blocks do not scale throughput forever. In the limit, very large blocks increase latency to a point where further throughput may not be gained. The long-version HotStuff paper [42] uses this technique, whose evaluation section shows the throughput saturates at batching hundreds of transactions ("400 vs. 800" curves). The sweet spot is adhoc to the specific application and its transactions, varying across practical blockchain instantiations and their deployment.

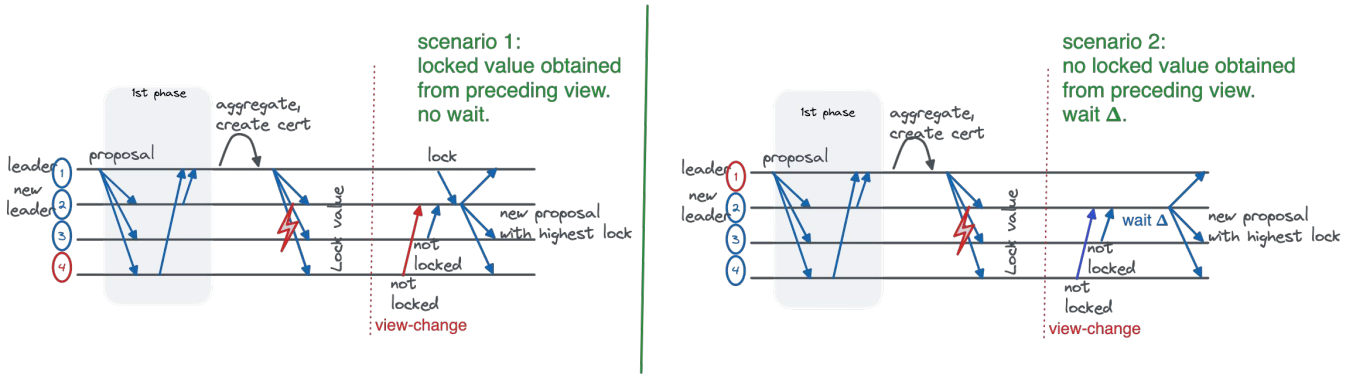

Figure 6: HotStuff-2: Some parties may not commit in a view but they become locked in it. Case 1: the highest lock is obtained by the next view leader and it proceeds responsively (left). Case 2: no honest party obtains a lock in a view, and the next view leader has to wait to propose in the next view (right).

Block Waves. Recently, an approach built on a different consensus foundation has demonstrated excellent resource utilization by nodes working in parallel on proposing/parsing blocks, and then driving a consensus decision on a *wave* of blocks. It is much more effective than batching because nodes can "buffer" blocks collaboratively and then let a consensus decision commit the entire wave. Moreover, information can continue spreading by nodes in the background while driving the next consensus decision, so that even if consensus stalls, the network continues having utility. More specifically, the idea is to let the entire network propose new blocks and organize the blocks by a layered DAG where each layer corresponds to a logical phase of the consensus protocol [12, 37]. Then, by some deterministic graph traversal, the blocks of each wave could be pipelined to commit in a linear order, triggered by the key phases. The upper diagram of Figure 7 sketches this approach in terms of network scheduling.

It is interesting to contrast the DAG approach with a "smart mempool" approach depicted in the bottom diagram. The idea is that blocks can be proposed in parallel and disseminated to the mempool with causal relations. Leaders can inject special blocks into the mempool, forming "bundles" in their proposals, and carrying QCs for previous bundles. The main difference is that bundles can have free structures, as shown in the figure. This is applicable to HotStuff and other chain-style protocols in general.

Concurrent Instances. Instead of carrying parallel work with the effort of a single leader, one can run multiple consensus instances concurrently, aka a "leaderless" approach, as in [19, 24, 38, 39]. The core idea in these protocols is to partition the replicated chain (log) according to some rules (e.g., round-robin) into pre-designated slots. All instances are performed in parallel by the nodes. Of course a realistic scheme needs to be fault tolerant, hence it needs a mechanism to handle faulty instances. This requires making a consensus decision, but the consensus method for this does not need to be high-throughput. Like the wave approach, the main drawback of running concurrent instances is the increased latency to reach finality.

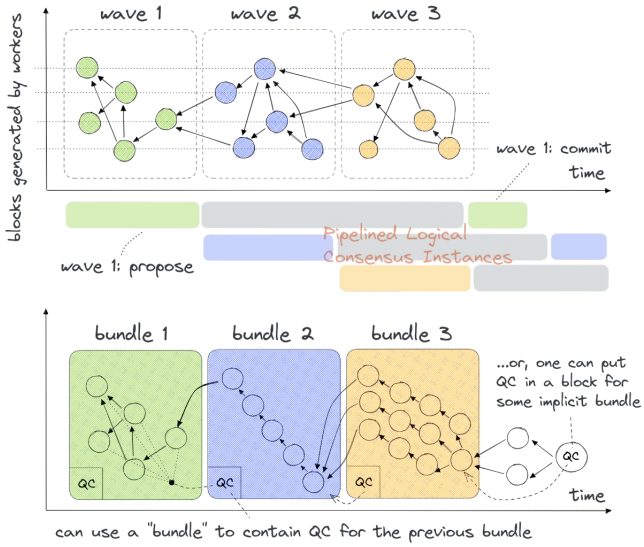

Figure 7: Driving waves/bundles of blocks.

Sharding. Since consensus offers fault-tolerance by introducing redundancy, scaling state-machine replication is fundamentally capped at the throughput of a single node. Therefore, the best scenario is that the replicas can perform as a cohort close to a single machine (the conceptual state machine being replicated) performance. It is worth remarking that some blockchain projects "scale out" via sharding [41], but this trades off fault tolerance, effectively reducing the global resilience down to the resilience of each single shard. Sharding is left out of scope from this short paper.

6.3 Concluding Remarks

We call on a systematic evaluation of the existing or emerging consensus systems, by clearly identifying the improvements brought by any of the aforementioned techniques and their combinations. Specifically, while the sequential logic in a consensus instance is inevitable, one can offload as much as possible from the core logic

so it is only left with lightweight small state mutation that is just enough to bookkeep the protocol state, and then parallelize work on the rest. Another important topic is separating data dissemination and availability from sequencing digests of the data. An additional issue is that end users usually do not directly participate in the consensus protocol, and thus the mempool used for disseminating user requests could create fairness issues with respect to sequencing, known as Miner/Maximal Extractable Value (MEV).

However, the common practice is to merely show full-system performance results and compare them against other full systems, which are also complex. In our experience, various engineering optimizations and system considerations may have surprising performance gains that have little to do with the fundamental consensus protocol. Moreover, common optimizations like batching and parallelizing message (signature) validation are applicable to many protocols. To avoid making apple-to-orange comparisons, the scientific community would benefit from a systematic, ingredient-by-ingredient study of the performance. Improving throughput, for example, affects the latency and it would be useful to know where it crosses a prohibitive point. Careful engineering is another point which would be beneficial to isolate.

Ultimately, to arrive at a high-performance, carefully engineered system, it requires using multiple techniques to saturate both the network and computational resources as much as possible.

On the foundational side, additional effort is needed to improve Pacemakers: the holy grail is a Pacemaker with expected linear communication, worst-case quadratic communication, and only $O(\Delta)$ delay per leader failure. The introduction of HotStuff-2 opens a door for a new generation of protocols. For example, it would be interesting to explore merging methods that were previously introduced to improve latency in HotStuff (e.g., [15, 16, 18]) into HotStuff-2. Another potential direction would be exploring if HotStuff-2 brings new insights or improvements in other fault models, e.g., in Momose-Ren [31] where the core structure of HotStuff is adapted to the Sleepy model of Pass and Shi [34].

ACKNOWLEDGMENTS

We are grateful to multiple projects that adopted HotStuff and shared their insights and improvements with us, and for excellent input that helped improve this manuscript by Kartik Nayak, Mike Reiter, and Alberto Sonnino.

REFERENCES

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. arXiv:cs.DC/1712.01367

[2] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019).

[3] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. 2020. Malicious security comes for free in consensus with leaders. *Cryptology ePrint Archive* (2020).

[4] Marcos Kawazoe Aguilera and Sam Toueg. 1999. A simple bivalency proof that t-resilient consensus requires t+ 1 rounds. *Inform. Process. Lett.* 71, 3-4 (1999), 155–158.

[5] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.* 75 (1987), 130–143.

[6] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph.D. Dissertation. University of Guelph.

[7] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *arXiv preprint arXiv:1710.09437* (2017).

[8] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. *IACR Cryptol. ePrint Arch.* (2001), 6. http://eprint.iacr.org/2001/006

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

[10] 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 *International Symposium on Distributed Computing*.

[11] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, 153–168.

[12] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 34–50. https://doi.org/10.1145/3492321.3519594

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

[14] Eli Gafni. 1998. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing (PODC '98)*.

[15] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022*. Springer, 296–315.

[16] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-commit proofs: Defeating livelock in BFT. *Cryptology ePrint Archive* (2021).

[17] 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 *Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019 (Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019)*. 568–580.

[18] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. 2020. Fast-HotStuff: A fast and resilient HotStuff protocol. *arXiv preprint arXiv:2010.11454* (2020).

[19] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2023. Practical View-Change-Less Protocol through Rapid View Synchronization. arXiv:cs.DB/2302.02118

[20] Jonathan Katz and Chiu-Yuen Koo. 2009. On expected constant-round protocols for Byzantine agreement. *J. Comput. System Sci.* 75, 2 (2009), 91–112. https://doi.org/10.1016/j.jcss.2008.08.001

[21] 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 (SOSP '07)*. Association for Computing Machinery, 45–58.

[22] Klaus Kursawe and Victor Shoup. 2005. Optimistic Asynchronous Atomic Broadcast. In *Automata, Languages and Programming*. 204–215.

[23] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (jul 1982), 382–401.

[24] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. 2019. FairLedger: A Fair Blockchain Protocol for Financial Institutions. In *International Conference on Principles of Distributed Systems*.

[25] Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. *ArXiv* abs/2201.01107 (2022).

[26] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022).

[27] Dahlia Malkhi and Oded Naor. 2022. The Latest View on View Synchronization. *https://blog.chain.link/view-synchronization/* (2022).

[28] Dahlia Malkhi and Kartik Nayak. 2023. Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT. Cryptology ePrint Archive, Paper 2023/397. https://eprint.iacr.org/2023/397 https://eprint.iacr.org/2023/397.

[29] Jean-Philippe Martin and L. Alvisi. 2006. Fast Byzantine Consensus. *Dependable and Secure Computing, IEEE Transactions on* 3 (08 2006), 202– 215. https://doi.org/10.1109/TDSC.2006.35

[30] James Mickens. 2014. The Saddest Moment. *login Usenix Mag.* 39, 3 (2014). https://www.usenix.org/publications/login/june14/mickens

[31] Atsuki Momose and Ling Ren. 2022. Constant Latency in Sleepy Consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 2295–2308.

[32] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2019. Cogsworth: Byzantine View Synchronization. *https://arxiv.org/pdf/1909.05204.pdf* (2019).

[33] Oded Naor and Idit Keidar. 2020. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. *ArXiv* abs/2002.07539 (2020).

[34] Rafael Pass and Elaine Shi. 2017. The Sleepy Model of Consensus. In *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, 380–409.

[35] HariGovind V. Ramasamy and Christian Cachin. 2005. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05)*. 88–102.

[36] Michael K. Reiter. 1994. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS '94)*. Association for Computing Machinery, 68–80.

[37] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[38] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolic. 2022. [Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *J. Syst. Res.* 2, 1 (2022). https://doi.org/10.5070/sr32159278

[39] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolic. 2022. State machine replication scalability made simple. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 17–33. https://doi.org/10.1145/3492321.3519579

[40] The Diem Team. 2021. DiemBFT v4: State Machine Replication in the Diem Blockchain. (2021). https://developers.diem.com/docs/technical-papers/state-machine-replication-paper.

[41] Gavin Wood. 2023. Polkadot: vision for a heterogeneous multi-chain framework. Cryptology ePrint Archive, Paper 2023/397. https://eprint.iacr.org/2023/397 https://eprint.iacr.org/2023/397.

[42] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT Consensus in the Lens of Blockchain. *CoRR* abs/1803.05069 (2018). arXiv:1803.05069

[43] 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 (PODC '19)*. Association for Computing Machinery, 347–356.