The Distributed Computing Column

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

This month, in the Distributed Computing Column, Neil Giridharan surveys some exciting recent work on DAG BFT protocols. Over the last decade, Byzantine fault-tolerant (BFT) state machine replication has proved to be one of the most effective and robust general techniques for building a distributed service. Traditionally, such protocols were based around repeatedly executing Byzantine agreement, where servers agreed iteratively on the sequence of actions to take. More recently, however, there has been a new "DAG-based" approach where servers collectively construct a DAG that can then be ordered in a consistent manner. This new approach has led to significant performance improvements, and enabled a variety of new optimizations (e.g., decoupling data transfer from agreement).

In this survey, Neil Giridharan provides an overview of this new class of protocols. He begins by describing Bullshark, the first partially sychronous DAG BFT protocol; and he then shows how more recent protocols such as Shoal++, Sailfish, and Cordial Miners, Mysticeti, and Autobahn build on Bullshark to optimize performance.

Overall, this article provides a comprehensive overview of the state-of-the-art for DAG BFT protocols today, along with some nice insights into the subtleties involved in high performance state machine replication.

The Distributed Computing Column is particularly interested in contributions that propose interesting new directions and summarize important open problems in areas of interest. If you would like to write such a column, please contact me.

Recent Results in DAG BFT

Neil Giridharan
UC Berkeley

Abstract

Traditional Byzantine fault tolerant (BFT) state machine replication (SMR) protocols have a reputation of having poor performance despite decades of work in optimizing these protocols. Recently, a new family of protocols, DAG BFT, has been shown to have much higher throughput than traditional BFT protocols despite having higher communication complexity. This has attracted a lot of interest in improving DAG BFT protocols. This brief survey gives background on the foundational DAG BFT protocols and summarizes the techniques of state-of-the-art results. Finally, it concludes by highlighting avenues for future work.

1 Introduction

Byzantine fault tolerant (BFT) state machine replication (SMR) protocols implement the abstraction of a single totally ordered log of client requests. These protocols must handle malicious failures and network asynchrony, which can arbitrarily delay messages. BFT SMR is a core building block of many systems such as blockchains, distributed databases, and cloud infrastructure.

The most well-known and extensively studied BFT SMR protocols are in the *traditional BFT* [6, 14, 26, 18, 11, 13, 15] family. Traditional BFT protocols typically work by having a leader drive progress, while having a view change mechanism to replace leaders that are faulty or slow. These family of protocols have been extensively optimized to achieve optimality in theoretical metrics such as round and communication complexity. Despite this theoretical optimality, traditional BFT SMR protocols have shown to have poor performance in practice [12, 7, 23].

Recently, a new family of BFT SMR protocols emerged known as *DAG BFT* [16, 7, 5, 10, 23, 22, 3, 17, 4]. These protocols work by having all replicas build a DAG of data proposals, and then totally ordering this DAG according to some deterministic rule to reach a consistent state. DAG BFT is suboptimal in theoretical metrics like communication complexity. However, they have impressive practical performance, achieving 10x throughput compared to traditional BFT protocols [7]. This is in large part because they decouple data dissemination from agreement, alleviating a key throughput bottleneck of traditional BFT SMR.

These impressive performance results have attracted a lot of interest in DAG BFT from academics and industry practitioners alike. They have been adopted by several blockchain projects [5, 1, 10] and have been cited numerous times. Unfortunately, these protocols are not the easiest to understand. DAG BFT uses different techniques and arguments, making it difficult for those unfamiliar to develop a strong intuition for these protocols. Furthermore, it is also hard to compare DAG BFT protocols, and to understand the various trade-offs that each protocol makes. This makes it difficult to tweak and extend these protocols, which is often necessary when trying to deploy these protocols into production.

This paper is a survey on DAG BFT protocols in the partial synchrony model [9], as partial synchrony is the most widely used network model for BFT SMR. Our goals are twofold. We first hope to give an intuition of how DAG BFT works, so that unfamiliar readers can understand new DAG BFT protocols. Secondly, we aim to cover the various techniques of recent state-of-the-art DAG BFT protocols with the goal of illustrating the different trade-offs of each protocol.

We first give background on Bullshark [23, 24], the first partially synchronous DAG BFT protocol (Section §3). Understanding Bullshark is the key to understanding many of the newer DAG BFT protocols, which either directly use Bullshark in a black-box manner or modify its internals. Next, we begin to cover recent DAG BFT results starting with protocols in the *certified DAG* family (Section §4). These protocols utilize techniques that do not modify or change the Bullshark's DAG structure. After that, we cover protocols in the *uncertified DAG* family (Section §5). These protocols use a similarly structured DAG but with different properties. Finally, we cover protocols in the *hybrid DAG* family (Section §6, which combine characteristics of traditional leader-based BFT protocols with DAG BFT.

2 Preliminaries

All covered protocols adopt the most popular BFT system model [6]. We assume a total of $n = 3f + 1$ replicas, where at most f of the n replicas can be faulty. Faulty replicas can behave arbitrarily, while correct replicas must execute the protocol faithfully. We assume a static adversary that can coordinate the actions of faulty replicas. Partial synchrony [9] is the adopted network model. In partial synchrony, there exists an unknown Global Stabilization Time (GST) and a known upper bound Δ on the message delivery time, such that before GST the network is asynchronous (no bound on message delivery time), but after GST all messages are guaranteed to arrive within Δ. We also often refer to δ as the *ac-*

tual message delay when analyzing latency. We assume standard cryptographic primitives like hash functions and digital signatures and assume that they cannot be broken by the adversary. Replicas use authenticated, reliable, point-to-point channels to communicate. They are assumed to check the validity of signatures before processing the corresponding messages.

BFT SMR outputs a linearizable totally ordered log of client requests which satisfy the following properties:

- **Safety.** No two correct replicas commit different values at the same log position.
- **Liveness.** All client requests are eventually committed in the log.
- **External Validity.** Any committed value must satisfy an external predicate P.

3 Bullshark

In this section, we give background on the Bullshark [23, 24] protocol, since it was the first partially synchronous DAG BFT protocol. Bullshark consists of two core components: (i) an algorithm that constructs a DAG of data proposals, and (ii) an algorithm that outputs a consistent total order of the DAG. We first focus on the algorithm that constructs the DAG (§3.1). We then show how the ordering algorithm (§3.2) totally orders the DAG without exchanging any additional messages.

3.1 Constructing the DAG

The DAG is divided into a number of integer-valued *rounds* (starting from 0), where in each round there is at most one vertex per replica. A *vertex* in the DAG represents a data proposal from a source replica, while an *edge* indicates a causal relationship between vertices. A valid vertex in round r only contains edges to at least $n-f$ vertices from round $r-1$. The *causal history* of a vertex, v, is the set of vertices that are reachable from v by following some sequence of edges. In every even-numbered round there is a pre-defined leader. An *anchor* is a vertex whose source replica is the leader for the round. The ordering algorithm uses anchors to reach a consistent total order.

DAG properties. The ordering algorithm requires the DAG to satisfy the following properties:

- **Non-equivocation.** For any position in the DAG, no two correct replicas will have different vertices.
- **Data availability.** For any vertex in a correct replica's DAG, at least one correct replica has stored the corresponding data proposal.

A DAG that satisfies these properties is known as a *certified* DAG.

Adding Vertices to the DAG. To add a vertex to the DAG, a replica invokes *consistent broadcast* on its data proposal. Consistent broadcast guarantees the following properties:

- **Validity.** If the source replica is honest, then all correct replicas will deliver the source replica's vertex.
- **Consistency.** If a correct replica delivers vertex v and another correct replica delivers vertex v', then $v = v'$.
- **Integrity.** A correct replica delivers at most vertex.

Consistency ensures the non-equivocation property of the DAG, while validity and integrity ensure that each DAG round contains at least $n - f$ vertices — one per replica. When a replica delivers a vertex from consistent broadcast it obtains a certificate, which acts as a proof of delivery and a proof of data availability. Then it adds the certificate to its local DAG, so that its next vertex can include at least $n - f$ certificates.

Consistent Broadcast Protocol. When a replica, r_i has a new vertex, v_i, containing a data proposal and $n - f$ certificates (edges) it sends a signed Propose message containing v_i to all replicas. Upon receiving a Propose message from a replica, r_i, a replica r_j checks (i) that the vertex contains $n - f$ certificates form the previous round and (ii) that it has not voted for another vertex in the same round from the same source replica. If these checks pass, a replica stores the associated data proposal, and sends a signed Vote message containing a hash of the vertex to replica r_i. When replica r_i receives $n - f$ matching Vote messages, it assembles them into a certificate, acting as a proof of delivery. Replica r_i then sends this certificate to all replicas. Once a replica receives this certificate it delivers vertex v_i and adds the certificate to its DAG.

Advancing DAG rounds. Once a replica has delivered $n - f$ vertices in round r, it has enough information to create a new vertex for round $r + 1$. However, eagerly doing so can compromise liveness of the ordering algorithm, which relies on hearing from slow but correct replicas. As a result, a replica starts a timer in every DAG round to wait for vertices from slow but correct replicas.

In even rounds, a replica waits to hear from the leader. It can move to the next round if it delivers the anchor or if the timer expires. For odd rounds, a replica can advance if the ordering algorithm commits the anchor in the previous round or if the timer expires. As an optimization (in odd rounds), replicas can also advance if they conclude that it was impossible for the anchor to be committed.

3.2 Totally Ordering the DAG

In traditional state machine replication or atomic broadcast protocols, replicas agree on an ordered sequence of values. Bullshark takes a similar approach by agreeing on a sequence of anchors. Once a sequence of anchors is committed, the causal history of each anchor in the sequence is ordered by any deterministic rule, to establish a consistent total order for all vertices in the DAG. Thus, the main goal of the ordering algorithm is to agree on a sequence of anchors. An example execution of the ordering algorithm is shown in Figure 1.

The sequence of anchors can be thought of a hash chain, where each anchor has a parent anchor. The parent of an anchor, A_i, is defined as the anchor with the highest round in A_i's casual history. An anchor A_j extends another anchor A_i if A_j is a descendant of A_i in the hash chain. Given an anchor, A, the sequence of anchors is determined by recursively computing the parent for each anchor. Once a replica commits an anchor, all prior anchors in the hash chain are also committed. To preserve safety, the commit rule needs to satisfy the following invariant.

- **Extension.** If a correct replica commits an anchor, A, in round r, then any anchor in round $r' > r$ must extend A.

The extension invariant ensures safety as any committed anchor must be in the prefix of any later anchor that is committed. For example, if one correct replica commits anchor A_1 and another correct replica does not (due to asynchrony or faults), then once the other correct replica commits a future anchor, A_2, A_2 is guaranteed to extend A_1; thus, A_1 will also be committed.

Commit Rule. The commit rule counts the number of votes for an anchor, where a vote is a vertex that has an edge to the anchor. A replica can commit an anchor, A, in round r if it observes $f + 1$ vertices (votes) in round $r + 1$ that each have an edge to A. This guarantees that any vertex in round $r + 2$ will have an edge to at least one of the $f + 1$ votes (by quorum intersection). The extension property is satisfied, since any future anchor is in round $r + 2$ or contains some vertex in round $r + 2$, which must have a path to the anchor in round r. Note that the commit threshold only needs to be $f + 1$ rather than $2f + 1$, since the DAG guarantees non-equivocation.

Final Total Order. Recall that an anchor, A, defines a hash chain of anchors where A is the tip. Once an anchor is committed, all anchors in the hash chain are also committed. To compute the anchor sequence, a replica recursively gets the parent anchor, which is the latest anchor in its causal history.

Once the sequence of anchors is computed, a replica will iterate through each anchor in order and execute its causal history in some deterministic order. This is safe since the causal history of an anchor is unique.

3.3 Analysis

Communication Complexity. Each vertex has size $O(n)$, since it contains $n - f$ edges. Each edge is a certificate containing $O(n)$ signatures, but assuming threshold signatures this cost can be reduce to $O(1)$. Consistent broadcast has $O(n^2)$ complexity for messages of size $O(n)$. So since there are $O(n)$ replicas each broadcasting a vertex, the total complexity is $O(n^3)$.

Throughput. Communication complexity has often been assumed to negatively correlate with throughput. However, prior work [7] observed that in practice data dissemination is a major throughput bottleneck. Bullshark parallelizes data dissemination across all replicas, avoiding a bottleneck at a single leader replica unlike other traditional BFT protocols. This results in better bandwidth utilization and higher throughput compared to traditional BFT protocols [7, 23], despite the higher communication complexity. In practice with batching, consensus metadata messages are often much smaller in size compared to data proposals. Thus, the overhead from more metadata messages does not significantly impact throughput.

Latency. We analyze latency under good-case conditions when the network is synchronous, and there are no faults. Anchors take 2 DAG rounds to commit, where each DAG round takes 3δ (message delays) due to consistent broadcast. Vertices in odd (non-anchor) rounds require an additional DAG round to be committed, since they must wait to be included by the anchor in the next round. Finally, vertices in even (anchor) rounds that are not the anchor require 4 DAG

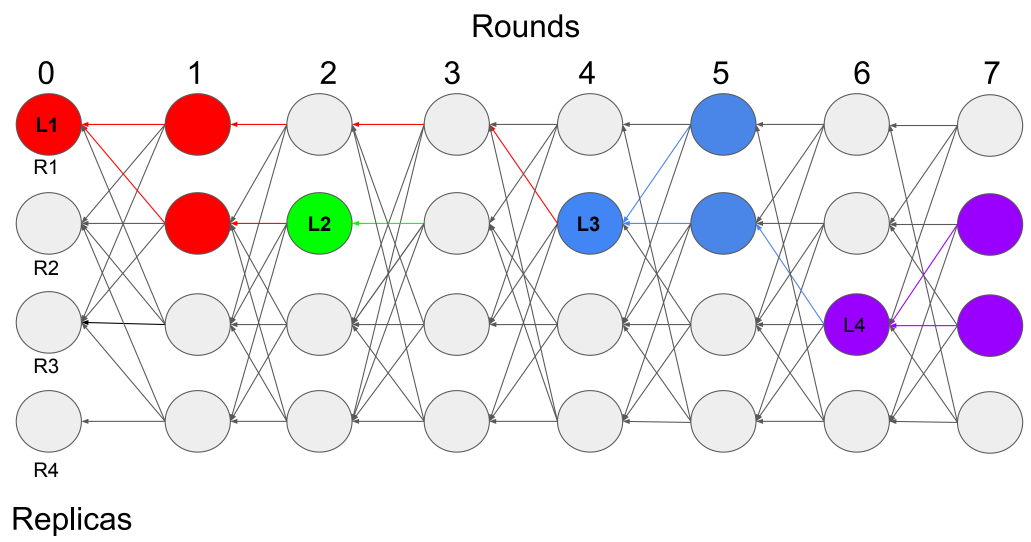

Figure 1: Illustration of the DAG at replica R1 with $n = 4, f = 1$. The columns represent the round numbers and the rows are all the vertices from a particular source replica. $L1$ denotes the first anchor (in round 0). There are $f + 1 = 2$ votes for $L1$, so $L1$ is committed. $L2$ is the second anchor in round 2. $L1$ is the parent of $L2$ in the hash chain sequence since it is the latest anchor in $L2$'s causal history. $L2$ only has one vote for it, so it is not committed. The next anchor, $L3$, does not have $L2$ in its causal history, but does have a path to $L1$, so $L1$ is its parent anchor. There are 2 votes for $L3$ in round 5, so the hash chain sequence $L1$, $L3$ is committed. Anchor $L4$ has $L3$ in its causal history, so $L3$ is its parent anchor. Similarly, it has 2 votes and so the hash chain anchor sequence of $L1, L3, L4$ is committed.

rounds to commit, since they must wait until the next anchor (in the next even round) includes them.

In summary, for an even rounds the anchor has a commit latency of 6δ, while non-anchor vertices have a commit latency 12δ. For odd rounds, vertices have a commit latency of 9δ. Thus, the expected commit latency is 10.5δ. However, the full consensus latency encapsulates not just the commit latency but also the queuing latency for a client request to be included in a vertex. Since a vertex is disseminated every DAG round, and each DAG round takes 3δ, on average it will take 1.5δ for a request to be included in a vertex. Thus, the full consensus latency is 12δ.

Chain Quality. In traditional BFT protocols there is no bound on the number of committed proposals from Byzantine replicas. However, in Bullshark, each DAG round contains at least $n - f$ vertices, of which at least a majority are from correct replicas. This guarantees a property called *chain-quality* which states that at least $\frac{1}{2}$ of committed blocks must be from correct replicas.

Fairness. The verison of Bullshark described here lacks fairness in the sense that vertices from some correct replicas may be "orphaned". Since each vertex is only guaranteed to have $n - f$ edges, vertices from slow but correct replicas in round r may not be referenced by vertices in round $r + 1$. Once this happens, the orphaned vertices from round r are effectively abandoned since no future vertex will reference them. To solve this issue, Bullshark adopts weak edges [16] to allow vertices to also have edges to some vertices not in the immediate prior round. Weak edges make garbage collection difficult, since vertices can reference prior vertices in arbitrarily lower rounds. We omit discussing how to do garbage collection while preserving fairness and refer to the Bullshark paper [23] for more details.

Data Synchronization. For a given vertex, v, a replica may only have a certificate proving data availability, and not the actual corresponding data payload for v or for vertices in v's causal history. A replica therefore must synchronize with other replicas to fetch all payload data if it does not already have the data locally. This data synchronization process unfortunately is sequential and may require multiple network round-trips. To see why, take an example of a vertex v in round 3. First, a replica must fetch the data for v's edges in round 2. Only when the vertices from round 2 are fetched does a replica know what vertices to synchronize on from round 1. Similarly, once all data for the vertices in round 1 are fetched, the replica must synchronize on the data for vertices in round 0.

Key question. Of the metrics we analyzed, latency is the major one for which Bullshark is suboptimal compared to other BFT protocols. Bullshark's consensus latency of 12δ is significantly higher than the optimal 3δ that leader-based protocols can achieve [2]. Thus, most recent work address the key question of whether it is possible to design a DAG BFT protocol that achieves low latency. Recent work has answered this question in the affirmative, using different designs. We classify the recent results into three different protocol families: Certified DAG, Uncertified DAG, and Hybrid DAG, and start by discussing the Certified DAG family.

4 Certified DAG

In this section, we cover the Shoal [22], Shoal++ [3], and Sailfish [21] protocols. These protocols belong to the certified DAG family. The certified DAG family of protocols adopt the same DAG as in Bullshark but adopt various techniques to reduce Bullshark's latency.

4.1 Shoal

Shoal [22] does not modify Bullshark internals, but instead applies two general-purpose techniques to reduce latency: pipelining and leader reputation. Shoal takes as a starting point a single-shot version of the Bullshark protocol. Single-shot Bullshark runs on a certified DAG starting at some round r. It terminates immediately upon committing the first anchor in some round $r' \geq r$.

Pipelining. Shoal creates a pipelining effect by invoking multiple single-shot Bullshark instances sequentially. Once an anchor is committed in round r, a new instance is started in round $r + 1$. Optimistically, every instance will commit an anchor in the first round it was invoked, resulting in an anchor being committed in every round. As in Bullshark, once an anchor is committed its causal history is ordered in some deterministic way. Safety is preserved since each instance inherits Bullshark's safety property, which ensures all correct replicas will agree on the same first anchor to order. Bullshark's safety property also ensures all correct replicas will agree on which round to invoke each new single-shot Bullshark instance.

Leader reputation. The pipelining effect can be nullified by faulty or slow leaders. To see this, suppose the first single-shot Bullshark instance commits an anchor in round 0 but the second instance commits an anchor in round 3 instead of round 1 because the round 1 leader was faulty. Vertices in rounds 1 and 2 experience high latency because they have to wait for the anchor in round 3 to be committed. To mitigate this issue, replicas can assign reputation scores to other replicas, and only elect leaders with high reputation scores. These reputation scores can be computed based on past participation history. A key challenge is ensuring that all correct replicas agree on reputation scores; otherwise, different replicas can elect different leaders when running a single-shot Bullshark instance, causing a possible safety violation. Shoal solves this challenge by computing reputation scores deterministically using only the causal history of the last committed anchor. Since all correct replicas will agree on the same first anchor to commit, and use the same deterministic function, they will compute the same reputation scores, and thus compute the same leaders to use for the next single-shot Bullshark instance.

4.2 Shoal++

Shoal still has the same commit latency for anchors as Bullshark (6δ). Shoal++ [3] proposes a faster commit rule to reduce this latency down to 4δ. Additionally, Shoal++ more aggressively pipelines to further reduce consensus latency.

Fast Commit Rule. Recall that the Bullshark commit rule requires $f + 1$ votes delivered through consistent broadcast. Shoal++ makes the observation that receiving $2f + 1$ Propose messages for vertices in round $r + 1$ that have an edge to the anchor in round r is enough to commit the anchor in round r (without needing to wait for consistent broadcast to finish for those vertices). To see why, note that of these $2f + 1$ messages at least $f + 1$ are from correct replicas. By the validity property of consistent broadcast, all correct replicas are guaranteed to eventually deliver the vertex that appears in a Propose message from a correct source replica. Thus, since there are at least $f + 1$ votes that will eventually be delivered for the anchor, it is safe to commit without waiting for consistent broadcast to complete. This saves two message delays compared to Bullshark's commit rule, reducing the anchor commit latency to 4δ.

Aggressive Pipelining. Shoal limited pipelining to at most one new single-shot Bullshark instance per round. Shoal++ supports pipelining multiple single-shot instances in the same round as in Mysticeti [4] (covered in Section §5.2). Shoal++ pipelines up to n instances in the same round. Each instance is assigned a monotonically increasing sequence number (starting from 1) corresponding to a position in a totally ordered log as in PBFT [6].

Unlike in Shoal, each single-shot instance is initialized with a round r **and** a source replica i. Replica i's vertex in round r acts as the first anchor in the single-shot instance. The instance either decides to commit replica i's vertex in round r or to skip it. Once a correct replica has committed or skipped an instance with sequence number j, it waits until all sequence numbers from 1 through $j - 1$ have been committed or skipped. It then iterates through each sequence number in order and either executes the causal history of the committed vertex in some deterministic order or skips the instance. By the safety property of Bullshark, all correct replicas will agree on the same committed vertex or all agree to skip, so a consistent total order is established. This aggressive pipelining essentially makes all vertices anchors in some instance, and so with the fast commit rule they experience a commit latency of 4δ.

More DAGs. Shoal++ additionally pipelines more DAGs to reduce the queuing delay for a client request to be included in a vertex. Normally a client request would take on average 1.5δ to be included in a vertex, since consistent broadcast completes in 3δ. Shoal++ staggers three DAGs in parallel, that are each offset by one message delay, so that for a given replica, a vertex is proposed every message delay in some DAG. Therefore, a new client request can be included in the next vertex of whichever DAG is ready, reducing the queuing delay to 0.5δ. To maintain a consistent total order, the committed anchor sequence for each DAG is interleaved into a single totally ordered log, where the causal history of each anchor is executed in some deterministic order.

4.3 Sailfish

Unfortunately, the fast commit rule is not always guaranteed to trigger in good-case conditions when the network is synchronous and there are no faults. Sailfish [21] addresses this issue by introducing a notion of timeout certificates to the protocol. Timeout certificates prevent situations where Byzantine replicas can cause correct replicas to skip voting. Timeout certificates additionally enable Sailfish to support an anchor in every round without pipelining. Sailfish also has a multi-leader variant, which pipelines multiple instances in a round.

Fast Commit Rule Revisited. Recall that the fast commit rule requires $2f + 1$ PROPOSE messages for vertices in round $r + 1$ that reference the anchor in round r. Suppose a Byzantine replica observes f Byzantine vertices that do not reference the anchor, and $f + 1$ correct vertices in round $r + 1$ that vote for the anchor in its DAG. This Byzantine replica can quickly propose a valid vertex in round $r + 2$ causing the f remaining correct replicas to jump to round $r + 2$. Since these f correct replicas will not vote in round $r + 1$, the normal Bullshark commit rule must be used since there are only $f + 1$ correct votes.

Timeout Certificates. The key problem is that a Byzantine replica can cause a subset of correct replicas to advance rounds too quickly and not vote. To solve this, Sailfish adds additional constraints to vertices so that Byzantine replicas cannot advance rounds too quickly. A valid vertex must now contain an edge to the anchor or a timeout certificate. A timeout certificate contains $2f + 1$ timeout messages, where a timeout message indicates that a replica did not receive the anchor after a certain amount of time. The previous attack fails with this change because if a Byzantine replica tries to propose a vertex without an edge to the anchor, then it must include a timeout certificate, which requires at least $f + 1$ correct replicas to timeout. Since correct replicas do not timeout in good-case conditions, no timeout certificate can form and the fast commit rule will be satisfied at every correct replica.

This constraint on the vertices also enables an anchor in every round without pipelining. If an anchor in round $r + 1$ has an edge to the anchor in round r, then the extension invariant is satisfied. Otherwise, if the anchor in round $r + 1$ has a timeout certificate, then it is not possible for the anchor in round r to have committed, so it is safe for the anchor in round $r + 1$ to not reference the anchor in round r.

4.4 Discussion

Shoal, Shoal++, and Sailfish utilize the same certified DAG backbone so they inherit the same chain quality, fairness, data synchronization, and throughput properties.

Communication Complexity. The communication complexity of Shoal and Shoal++ remains at $O(n^3)$, since they do not add any extra messages. Sailfish also has the same complexity because timeout certificates can be reduced to $O(1)$ size using threshold signatures.

Latency. In terms of latency, Shoal still requires two DAG rounds or 6δ to commit anchors. However, with pipelining there is an anchor in every round, so non-anchor vertices only have a latency of 9δ. The queuing latency is on average 1.5δ since vertices are proposed every 3δ. The full consensus latency is thus 10.5δ.

Shoal++ reduces the commit latency of anchors to 4δ, and pipelines aggressively so that all vertices experience 4δ commit latency. By pipelining DAGs, Shoal++ also reduces the queuing latency down to 0.5δ for a total end-to-end latency of 4.5δ. Sailfish also uses the fast commit rule, so anchor commit latency is also 4δ. Non-anchor vertices require an extra DAG round to be included by an anchor, resulting in a commit latency of 7δ. However, applying the same pipelining optimizations to Sailfish results in a consensus latency of 4.5δ.

5 Uncertified DAG

The uncertified DAG family of protocols assume a DAG structure similar to a certified DAG. However, they do **not** require the DAG to satisfy the non-equivocation property. As a result, they eschew consistently broadcasting vertices and instead send vertices on a best-effort basis. Best-effort broadcast reduces latency by two message delays compared to consistent broadcast but requires the ordering algorithm to have additional complexity to handle possibly inconsistent DAGs. We first present Cordial Miners [17], and then cover Mysticeti [4].

5.1 Cordial Miners

As with the certified DAG, the uncertified DAG is also divided into a number of integer-valued rounds. In each round, a source replica attempts to add a vertex to the DAG, where each vertex contains $n - f$ edges to vertices in the previous round. However, since replicas do not consistently broadcast vertices, it is possible for a Byzantine replica to equivocate and have multiple vertices in a given round.

DAG Structure. Cordial Miners logically groups every three rounds in the DAG into a *wave*. For example, rounds 0, 1, and 2 are in wave 1 rounds 3, 4, and 5 are in wave 2, etc. In the first round of a wave there is a pre-defined leader, and its associated vertex is the anchor. Note that if the leader is Byzantine, there may be multiple anchors due to equivocation. The second round in the wave is a voting round, where a vote is a vertex in round $r + 1$ which has an edge to an anchor in round r. The third round of the wave is the final voting round, where if enough votes are observed an anchor is committed.

Adding vertices to the DAG. When a replica wants to add a vertex to the DAG it sends a Propose message containing this vertex to all replicas. Upon receiving a Propose message with a new vertex, a replica adds the vertex to its local DAG **only** if it has received the entire causal history of the vertex. If replicas do not wait to have the entire causal history, then it is possible for a vertex to be committed without any correct replica having the entire causal history for that vertex. This violates the data availability property, which compromises liveness. If a replica is missing the causal history locally, it can ask the source replica to send over the missing vertices. Note that correct replicas may receive equivocating vertices. Although this is proof of Byzantine behavior, correct replicas still need to add these vertices to their DAGs because it is possible one of these vertices may be committed.

Advancing DAG rounds. As with certified DAGs, correct replicas must use timeouts in each DAG round to ensure liveness. Upon entering a new DAG round, replicas start a timer, and wait to receive at least $n - f$ vertices in the current round. Then, a replica checks whether the following conditions hold. If a replica is in the first round of a wave, it advances if it received an anchor or times out. If a replica is in the second round of a wave, it advances if it received a quorum of $n - f$ votes for an anchor or times out. Finally, if a replica is in the third round of a wave, it advances if it commits an anchor or times out.

Totally Ordering the DAG. As with the certified DAG family of protocols, the ordering algorithm agrees on a hash chain of anchors. Once this sequence of anchors is committed, the causal history of each anchor can be committed in some deterministic order. To maintain safety, the commit rule must satisfy the extension invariant, which ensures that if an anchor is committed, then any future anchor in the hash chain must extend it.

Commit Rule. An anchor, A, in round r can be safely committed if there are $n - f$ votes in round $r + 2$ that have edge to a certificate for A. A certificate for A consists of $n - f$ votes in round $r + 1$ for A. The certificate ensures non-equivocation in that at most one anchor in a wave can receive $n - f$ votes (by quorum intersection). The commit rule satisfies the extension invariant because any vertex in round $r + 3$ must have an edge to one of the certificate votes in round $r + 2$ (by quorum intersection). Since any future anchor must be in round $r + 3$ or have a vertex in round $r + 3$ in its causal history, the extension invariant holds.

Final Total Order. Like in certified DAGs, given an anchor, the full hash chain of anchors is defined by recursively computing the parent anchor. The parent, A_p, of an anchor, A, is the latest anchor (highest round) in A's causal history such that there is a vertex which references a certificate of A_p. A vertex which references a a certificate is when there is a vertex in A's causal history that has $n - f$ votes for A_p. Once an anchor is committed, the hash chain sequence of anchors is also committed. A replica will then iterate through each anchor in sequence order and execute the causal history of each anchor in some deterministic order.

5.2 Mysticeti

Mysticeti [4] uses the core Cordial Miners [17] protocol but adds aggressive pipelining similar to Shoal++ [3]. In each round, there are k proposer slots, which are totally ordered. A proposer slot runs an instance of a single-shot Cordial Miners. For a single-shot instance for a proposer replica i in round r, the first anchor is initialized to be a vertex from replica i in round r. Once a commit or skip decision is reached for this anchor, the single-shot instance terminates. Before a proposer slot is executed, a replica must wait for all prior proposer slots (in earlier rounds or smaller ranked slots in the same round) to either be committed or skipped. A replica then iterates through all committed slots in order and executes the causal history of each committed vertex in some determinsitic order.

5.3 Discussion

A benefit of Cordial Miners and Mysticeti is that they require fewer signatures compared to certified DAG protocols. This is because they do not require forming certificates containing $n - f$ signatures for each vertex. A single signature is effectively shared as a certifying vote for several vertices.

Data Synchronization. A drawback of Cordial Miners and Mysticeti is that replicas must fetch the entire causal history of a vertex before adding it to the DAG. Otherwise, replicas may commit a vertex v, where no correct replica has some vertex, v', in v's casual history. This can happen if Byzantine replicas do not send their vertices to all replicas. Unlike with certified DAGs, this data synchronization step must occur on the critical path of consensus. In the worst case, correct replicas may have to synchronize on an arbitrary large amount of data, which can trigger timeouts even under benign network conditions. Furthermore, this synchronization step increases the communication complexity to $O(n^4)$, as each correct replica may have to send a causal history of $O(n)$ vertices of size $O(n)$ to $O(n)$ replicas.

Latency. For latency, Cordial Miners requires 3 DAG rounds to commit an anchor, where a DAG round has δ latency. Non-anchor vertices in the first round of a wave require 6 DAG rounds to commit since they must wait for the next anchor to be committed. Vertices in the second and third round of a wave require 5 and 4 DAG rounds respectively as they must also wait for the next anchor to be committed. In total, the commit latency is 5δ. Since vertices are proposed every δ, the expected consensus latency is 5.5δ.

Mysticeti aggressively pipelines so that each vertex has the same commit latency as the anchor latency (3δ). Unfortunately, even with good-case conditions it is not guaranteed that the k slots in a round will all be committed. This requires the $n-f$ replicas who voted for the first slot in a round to also vote for all the other slots. When this condition does hold, Mysticeti has a consensus latency of 3.5δ (3δ commit latency and 0.5δ queuing delay). Note, though, that not every message delay is created equal. Each message delay in Cordial Miners and Mysticeti requires sending a full data proposal which in practice can take longer to transmit than consensus metadata messages. In contrast, with certified DAGs only the first message of consistent broadcast requires sending the full data proposal.

6 Hybrid DAG

Hybrid DAG protocols combine a DAG-based data dissemination layer with a traditional leader-based BFT consensus protocol to get the best of both worlds. Instead of relying on a commit rule that counts the number of edges that reference a particular vertex, hybrid DAG protocols use a traditional BFT consensus protocol to commit vertices. A traditional BFT consensus protocol commits vertices with low latency, while the DAG layer ensures high throughput by parallelizing data dissemination across all replicas. We first describe the Autobahn [12] system, and then cover Star [8] and BBCA-Chain [19].

6.1 Autobahn

In Autobahn [12], replicas construct a DAG of data proposals. This DAG has a simpler structure compared to the other DAG protocols, since the DAG edges are not used for consensus. Periodically, the consensus protocol will commit snapshots of the DAG. Once a snapshot has been committed, replicas must fetch the data proposals of the snapshot using a data synchronization protocol. Finally, replicas can execute all the vertices in the snapshot in any common deterministic order. We first describe how the DAG layer works, followed by the consensus layer, and then finally the data synchronization protocol.

DAG Layer. Unlike prior DAG protocols, a valid vertex from a source replica i has only one edge to the previous vertex from replica i. The resulting DAG consists of n parallel hash chains, one per replica.

Autobahn uses *weak consistent broadcast* to disseminate vertices. Weak consistent broadcast follows the same protocol as consistent broadcast but instead of a certificate containing $n - f$ Vote messages, it contains $f + 1$ Vote messages. Certificates in weak consistent broadcast do **not** protect against equivocation. Instead, they prove that at least one correct replica has stored the corresponding data proposal for the vertex. This ensures that if a correct replica commits a vertex, the corresponding data proposal is guaranteed to be retrievable.

A replica can propose a new vertex as soon as its last vertex was delivered through weak consistent broadcast. Unlike prior DAG protocols, replicas do not proceed in synchronized rounds or wait for timeouts. Replicas propose vertices at their own pace as soon as their own vertices have been delivered. Vertices can be pipelined to further reduce latency.

Consensus Layer. Replicas may have diverging DAGs due to equivocation or asynchrony. The consensus layer reconciles these differences by ensuring that replicas agree on common snapshots of the DAG. A snapshot consists of the latest vertex from each source replica (n certificates), representing the frontier of the DAG. To ensure that snapshots cover new vertices, correct replicas must propose snapshots that satisfy a coverage parameter. This coverage parameter indicates the number of hash chains that have at least one new vertex since the previous snapshot. It is typically set to $n - f$ (since at most f are faulty). The specific consensus protocol used is a slightly modified version of PBFT. However, in principle any consensus protocol can be used. Once a particular snapshot is committed, replicas fetch the causal history of all n vertices that represent the snapshot and commit them in some common deterministic order.

Data Synchronization. Autobahn takes advantage of the simpler DAG structure to have a faster data synchronization protocol. In Autobahn, a replica only votes for a vertex in weak consistent broadcast from a source replica once it has already voted for all prior vertices in that source replica's hash chain (causal history). Since replicas vote for vertices in order, if a vertex has $f + 1$ votes, then at least one correct replica from this set of $f + 1$ has stored the entire causal history. Thus, a replica can get the entire causal history of a vertex in just a single round-trip from asking the set of $f + 1$ replicas that voted for the vertex. This is in contrast to certified DAG protocols which must recursively synchronize, incurring multiple round-trips to fetch the entire causal history.

6.2 Star

Star [8] has a similar DAG layer and consensus layer, where the DAG layer consists of n parallel hash chains and the consensus layer uses a traditional BFT protocol like PBFT. The DAG layer also uses weak consistent broadcast to deliver vertices, but unlike Autobahn it does proceed in synchronized rounds. A replica can only proceed to the next round and propose a new vertex upon delivering $n - f$ vertices in the current round. The consensus layer runs a consensus instance per round, which agrees upon a set of at least $n - f$ vertices to output. Replicas can then order the committed vertices in some deterministic way. The DAG and consensus layers can also be pipelined to reduce latency.

6.3 BBCA-Chain

BBCA-Chain [20] uses an uncertified DAG but differs from Cordial Miners and Mysticeti in that it does not use a DAG-based commit rule for ordering. Instead, it leverages a traditional leader-based BFT protocol, BBCA, similar to PBFT, to agree on a hash chain of leader vertices. BBCA messages can be piggybacked onto the messages to construct the DAG, so that it has minimal overhead.

Leader and non-leader vertices both have $n-f$ edges to vertices in the previous DAG round; however, they differ in how they are disseminated. Non-leader vertices are proposed in every DAG round using best-effort broadcast, while leader vertices are proposed every time BBCA outputs a valid ticket. This ticket is typically a commit certificate for the previous leader vertex, but can also be a set of timeout messages that indicate a quorum of replicas have not committed the previous leader vertex. The ticket is necessary so that the extension property is satisfied. If a leader vertex v is committed, then any future leader vertex must have a ticket which extends v.

6.4 Discussion

Communication Complexity. The simplified DAG structure reduces the communication complexity of the data layer to $O(n^2)$ for Autobahn and Star since each vertex only has a single edge instead of $n - f$. While consensus inputs are of size $O(n)$, using a traditional BFT consensus protocol still retains $O(n^2)$ communication complexity. Autobahn's fast data synchronization may incur $O(n^3)$ in the worst case, as $O(n)$ replicas may need to synchronize with $O(n)$ replicas on $O(n)$ tips. BBCA-Chain still retains the uncertified DAG structure of Sailfish and Mysticeti, so its communication complexity is also $O(n^4)$.

Latency. Autobahn's consensus latency is 4.5δ applying aggressive pipelining and an optimization for consensus to propose uncertified vertices. This includes δ for a vertex to be disseminated, 0.5δ to be included in a consensus proposal, and another 3δ for the consensus protocol to commit the vertex. Star's consensus latency (including pipelining) includes 0.5δ for a request to be included in a vertex, 2δ for the vertex to be added to the DAG (weak consistent broadcast), and 3δ for consensus to terminate. For BBCA-Chain, vertices are proposed every δ, so on average the queuing delay is 0.5δ for a request to be included in a vertex. Leader vertices experience a latency of 3δ using BBCA. Non-leader vertices take δ to be disseminated by best-effort broadcast, 1.5δ to be included by the next leader vertex, and 3δ for BBCA to commit that vertex.

Chain Quality. Star achieves chain quality by ensuring that each consensus instance decides at least $n - f$ vertices, of which at least $n - 2f$ are from correct replicas ($\geq \frac{1}{2}$ of total replicas). Autobahn does not cap the amount of vertices that can be committed from a particular consensus instance, so it is possible that a majority of committed vertices are from Byzantine vertices. Autobahn thus sacrifices chain quality to provide each replica the same opportunity for progress.

Seamlessness. A core contribution of Autobahn is defining a property called seamlessness. Seamlessness is a property that attempts to capture the robustness of a partially synchronous protocol. In a truly robust system, a synchronous period should not be affected by asynchrony that occurred before. Specifically, seamlessness states that after a period of asynchrony there should not be a performance degradation that lasts once synchrony returns.

Traditional BFT protocols are not seamless since during asynchrony they may fail to make any progress causing a backlog of uncommitted requests to form. The following synchronous period must work off this entire backlog, causing latency to spike until this backlog is worked off completely. Certified DAG protocols are almost seamless, but not quite. During an asynchronous period, consensus may stall but vertices are continually added to the DAG. Once synchrony returns, and consensus commits a new vertex, these uncommitted vertices from the asynchronous period are also committed as part of the causal history. As a result, latency does not spike due to a large backlog. However, the data synchronization process to fetch the causal history of a committed vertex can take many round-trips, leading to a latency increase.

Uncertified DAGs (including BBCA-Chain) are not seamless, since data synchronization must occur on the critical path of consensus. After an asynchronous period, there could be a large amount of data to synchronize on, which can cause timeouts to fire, increasing latency. Star is also not seamless because during asynchrony a consensus instance is started per DAG round. Once synchrony returns, all these consensus instances need to terminate before new requests can be executed.

Autobahn is seamless since the DAG grows during asynchrony. Then, once a synchronous period returns the first consensus instance commits the entire sub-DAG that formed during asynchrony. The fast data synchronization protocol ensures that fetching the entire sub-DAG takes just a single round-trip.

7 Conclusion

This survey covered state-of-the-art results in partially synchronous DAG BFT. Significant progress has been made in reducing communication complexity and latency. However, recent results have not quite achieved optimal latency [2, 6]. Progress has also been made on practical issues such as improving fairness, censorship resistance, and data synchronization. One challenge in particular is achieving censorship resistance without committing duplicate transactions. Mir-BFT [25] proposed a solution, but further improvements would be interesting future work.

References

[1] Sui Blockchain. (last accessed on 09/23/24). URL: `https://sui.io/`.

[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 331–341, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3465084.3467899`.

[3] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput DAG BFT can be fast and robust! In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 813–826, Philadelphia, PA, April 2025. USENIX Association. URL: `https://www.usenix.org/conference/nsdi25/presentation/arun`.

[4] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

[5] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 34:9–11, 2016.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[7] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[8] Sisi Duan, Haibin Zhang, Xiao Sui, Baohan Huang, Changchun Mu, Gang Di, and Xiaoyun Wang. Dashing and star: Byzantine fault tolerance with weak certificates. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 250–264, 2024.

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

[10] Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes, 2019. URL: `https://arxiv.org/abs/1908.05156`, `arXiv:1908.05156`.

[11] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, page 296–315, Berlin, Heidelberg, 2022. Springer-Verlag. `doi:10.1007/978-3-031-18283-9_14`.

[12] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 1–23, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3694715.3695942.

[13] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. Beegees: stayin'alive in chained bft. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 233–243, 2023.

[14] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, USA, 2019. IEEE. doi:10.1109/DSN.2019.00063.

[15] Mohammad M. Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient hotstuff protocol. *CoRR*, abs/2010.11454, 2020. URL: https://arxiv.org/abs/2010.11454, arXiv:2010.11454.

[16] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467905.

[17] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[18] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)*, 27(4):7:1–7:39, January 2010. URL: http://doi.acm.org/10.1145/1658357.1658358, doi:10.1145/1658357.1658358.

[19] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: Low latency, high throughput bft consensus on a dag. *Financial Cryptography and Data Security (FC'24)*, 2024.

[20] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: Low latency, high throughput bft consensus on a dag. In Jeremy Clark and Elaine Shi, editors, *Financial Cryptography and Data Security*, pages 51–73, Cham, 2025. Springer Nature Switzerland.

[21] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. Sailfish: Towards Improving the Latency of DAG-based BFT . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1928–1946, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00021, doi:10.1109/SP61157.2025.00021.

[22] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *Financial Cryptography and Data Security (FC'24)*, 2024.

[23] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3559361.

[24] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633*, 2022.

[25] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-bft: High-throughput robust bft for decentralized networks. *arXiv preprint arXiv:1906.05552*, 2019.

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