Bullshark: The Partially Synchronous Version

Alexander Spiegelman[1], Neil Giridharan[2], Alberto Sonnino[3], and Lefteris Kokoris-Kogias[4]

[1]Aptos
[2]University of California, Berkeley
[3]Mysten Labs
[4]IST Austria

Abstract

The purpose of this manuscript is to describe the deterministic partially synchronous version of Bullshark in a simple and clean way. **This result is published in CCS 2022**, however, the description there is less clear because it uses the terminology of the full asynchronous Bullshark. The CCS version ties the description of the asynchronous and partially synchronous versions of Bullshark since it targets an academic audience. Due to the recent interest in DAG-based BFT protocols, we provide a separate and simple description of the partially synchronous version that targets a more general audience. We focus here on the DAG ordering logic. For more details about the asynchronous version, garbage collection, fairness, proofs, related work, evaluation, and efficient DAG implementation please refer to [8, 7, 3, 6]. For an intuitive extended summary please refer to the blogpost [1].

1 Introduction

In the context of Blockchains, BFT consensus is a problem in which n parties, f of which might be Byzantine, try to agree on an infinitely growing sequence of transactions. The idea of DAG-based BFT consensus is to separate the network communication layer from the ordering (consensus) logic. Each message contains a set of transactions, and a set of references to previous messages. Together, all the messages form a DAG that keeps growing – a message is a vertex and its references are edges. The networking layer of building the DAG can and should be optimized on a system level (see Narwhal [3]). Once a DAG is constructed the ordering logic of its vertices adds zero communication overhead. That is, each party independently looks at its local view of the DAG and totally (fully) orders all the vertices without sending a single extra message. This is done by interpreting the structure of the DAG as a consensus protocol, i.e., a vertex can be a proposal and an edge can be a vote. Importantly, due to the asynchronous nature of the network, different parties may see slightly different DAGs at any point in time. A key challenge then is how to guarantee that all the parties agree on the same total order.

2 Protocol

We describe here the partially synchronous version of Bullshark. To focus on the ordering logic of the DAG (Section 2.2), we first assume a DAG with certain properties 2.1 is given. Then, for completeness, we discuss a few alternatives to construct the DAG (Section 2.3). Details about the fairness, garbage collection, evaluation can be found in [8, 7, 1], and an efficient DAG implementation in [3]. Check [8] for rigorous (safety and liveness) proofs.

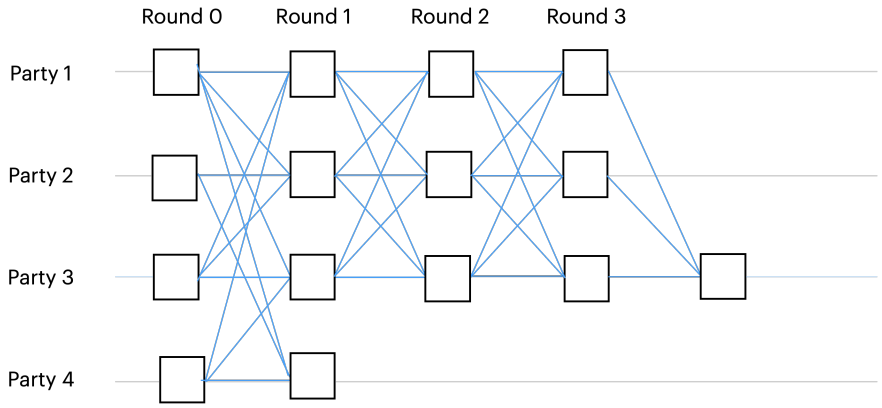

Figure 1: Illustration of the DAG structure.

2.1 DAG properties.

We consider a round-based DAG, see illustration in Figure 1. Each round contains at most n vertices (at most 1 vertex per party). Each vertex is associated with a *round* number and its *source* (the party that broadcast it). In addition to transactions information, each vertex also contains a set of at least $n - f$ *edges*, which point to vertices in the previous round.

We discuss the construction of the DAG below, but it is important to note now that due to the asynchronous nature of the network, different parties may see slightly different views of the DAG at any point in time. In order to guarantee that all the parties agree on the same total order of the DAG's vertices without any extra communication, the construction of the DAG must satisfy the following:

Validity: if an honest party has a vertex v in its local view of the DAG, then it also has all the causal history of v (all vertices that can be reached from v).

Reliability: if an honest party has a vertex in round r by party p in its local view of the DAG, then eventually all honest parties has a vertex in round r by party p in their views of the DAG.

Non-equivocation: if two honest parties have a vertex in round r by party p in their local views of the DAG, then the vertices are identical (i.e., transaction information and edges are exactly the same).

By recursively applying Validity and Non-equivocation we get:

Completeness: if two honest parties have a vertex v in round r by party p in their local views of the DAG, then v's causal histories are identical in both parties' local view of the DAG.

2.2 Ordering logic.

The non-equivocation property of the DAG eliminates the ability of Byzantine parties to lie, and as a result, the completeness property allows us to apply deterministic logic to order the DAG even though parties have slightly different local views of the DAG, which drastically simplifies the ordering logic.

As mentioned above each party observes its local view of the DAG and totally orders its vertices without any extra communication. **Notably, Bullshark needs neither a view-change nor a view-synchronization mechanism.** Since the DAG encodes full information, there is no need to "agree" on skipping and discarding slow/faulty leaders via timeout complaints and view-synchronization we get for free by the nature of the DAG construction.

We next demonstrate how each party locally interprets the structure of its view of the DAG via a running example with $n = 4$ and $f = 1$. A detailed pseudocode appears in Algorithms 1. and 2 (the function TryCommitting(v) is invoked by party p_i when a new vertex in an even round is added to its local view of the DAG).

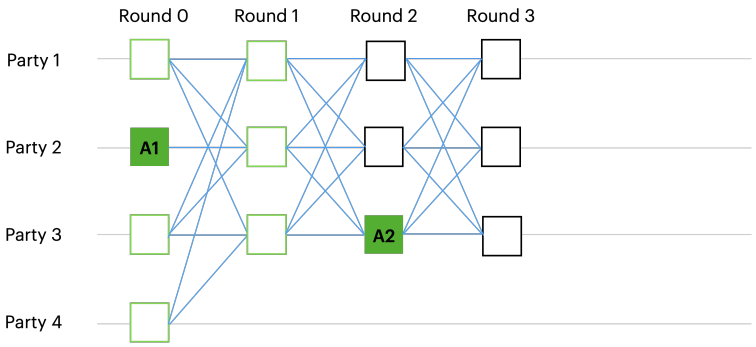

Figure 2: Anchors and causal history.

Every even-numbered round in the DAG has a predefined leader and we refer to the vertex associated with the leader as anchor. In figure 3, anchors are highlighted in solid green. The goal is to first decide which anchors to commit. Then, to totally order all the vertices in the DAG, a party goes one by one over all the committed anchors and orders their causal histories by some deterministic rule. Anchor $A2$ causal history is marked by green-outlined vertices.

Each vertex in an odd round can contribute one vote for the anchor in previous round. In particular, a vertex in round r votes for the anchor in round $\mathsf{r}-1$ if there is an edge between them. **The commit rule is simple:** an anchor is *committed* if it has at least $f+1$ votes. In Figure 3, anchor $A2$ is committed with 3 votes, whereas anchor $A1$ only has 1 vote and is not committed.

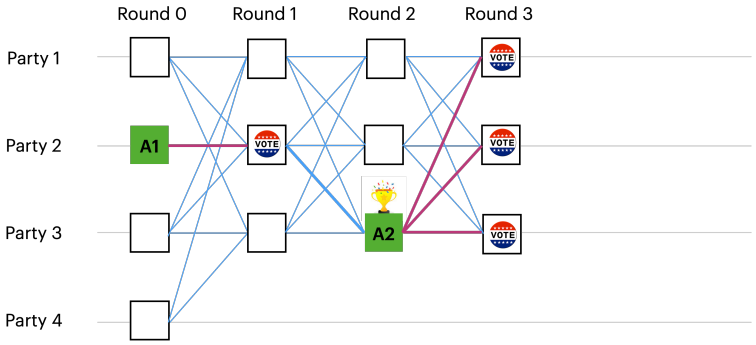

Figure 3: Commit rule requires $f+1$ votes.

Recall that due to the asynchronous nature of the network, the local views of the DAG might differ for different parties. That is, some vertices might be added to the local view of the DAG of some of the parties but not yet added by the others. Therefore, even though some parties have not committed A1, others might have. In Figure 4, party p_2 sees $2 = f+1$ votes for anchor $A1$ and thus commits it even though party p_1 has not. Therefore, to guarantee total order (safety), party p_1 has to order anchor $A1$ before anchor $A2$. To achieve this, Bullshark relies on quorum intersection:

> *Since the commit rule requires $f+1$ votes and each vertex in the DAG has at least $n-f$ edges to vertices from the previous round, it is guaranteed that if some party commits an anchor A then all anchors in higher rounds will have a path to at least one vertex that voted for A, and thus will have a path to A.*

Therefore, we get the following corollary:

> ***If there is no path to a anchor A from a future anchor, then no party committed A and it is safe to skip it.***

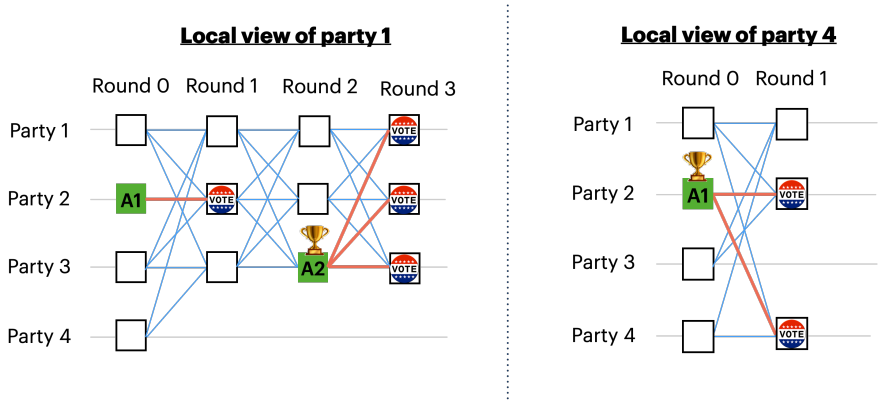

Figure 4: Parties might have slightly different views of the DAG.

The mechanism to order anchors is the following: when an anchor A_i is committed, the validator checks if there is a path between anchor A_i to A_{i-1}. If this is the case, anchor A_{i-1} is ordered before A_i and the mechanism is recursively restarted from A_{i-1}. Otherwise, anchor A_{i-1} is skipped and the validator checks if there is a path between A_i to A_{i-2}. If there is a path, A_{i-2} is ordered before A_i and the mechanism is recursively restarted from A_{i-2}. Otherwise, anchor A_{i-2} is skipped and the process continues in the same way. The process stops when it reaches an anchor that was previously ordered (as all the anchors before it are already ordered).

In Figure 5, anchors $A1$ and $A2$ do not have enough votes to be committed and once the party commits $A3$ it has to decide whether to order $A1$ and $A2$ before $A3$. Since there is no path from $A3$ to $A2$, $A2$ can be skipped (no party committed it). However, since there is a path between $A3$ and $A1$, $A1$ is ordered before $A3$). In this example the process stops here because $A1$ is the first anchor, but in general the process should continue recursively from $A1$ until a previously ordered anchor is reached. Note that it might be the case that no party committed $A1$, however it is safe to order it before $A3$ due to the Completeness property of the DAG. All parties see the same history of $A3$ in their local vies of the DAG and thus deterministically agree to order $A1$ before $A3$. In [8] we prove by induction that all parties order the same anchors.

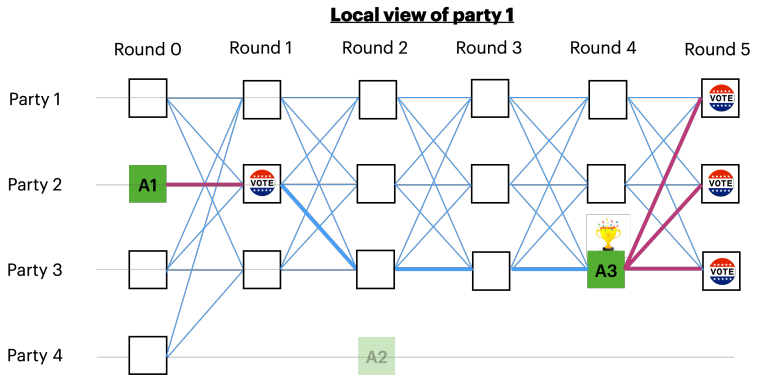

Figure 5: Anchors' ordering logic. $A2$ is not part of party 1 local view if the DAG

Finally, to totally order the vertices of the DAG, the party orders the causal history of the ordered anchors one by one by some deterministic rule. In our example, it first order $A1$ and then orders the causal history of $A3$. For an intuitive explanation of garbage collection and fairness, and for a DAG construction discussion please refer to the blogpost [1].

Algorithm 1 Data structures and basic utilities for party p_i

Local variables:
struct *vertex* v: ▷ The struct of a vertex in the DAG
v.*round* - the round of v in the DAG
v.*source* - the party that broadcast v
v.*block* - a block of transactions information
v.*edges* - a set of at least $n - f$ vertices in v.*round* $- 1$ ▷ Used to provide fairness
$DAG_i[]$ - An array of sets of vertices

procedure PATH(v, u) ▷ Check if exists a path from v to u in the DAG
  return exists a sequence of $\mathsf{k} \in \mathbb{N}$, vertices $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_\mathsf{k}$ s.t.
    $\mathsf{v}_1 = \mathsf{v}$, $\mathsf{v}_\mathsf{k} = \mathsf{u}$, and $\forall j \in [2..k]$: $\mathsf{v}_\mathsf{j} \in \bigcup_{\mathsf{r}\geq 1} \mathsf{DAG_i}[\mathsf{r}] \wedge \mathsf{v}_\mathsf{j} \in \mathsf{v}_{\mathsf{j}-1}.edges$

procedure GETANCHOR(r)
  p ← GETPREDEFINEDLEADER(r) ▷ Assume some predefined mapping known to all parties
  if $\exists$v ∈ DAG[r] s.t. v.*source* = p **then**
    return v
  return $\perp$

Algorithm 2 Eventually synchronous BullShark: algorithm for party p_i.

Local variables:
orderedVertices ← {}
lastOrderedRound ← 0
orderedAnchorsStack ← initialize empty stack

procedure TRYCOMMITTING(v)
  if v.*round* mod 2 = 1 or v.*round* = 0 **then**
    return
  anchor ← GETANCHOR(v.*round-2*)
  votes ← v.*edges*
  if $|\{\mathsf{v}' \in \mathsf{votes} : \text{PATH}(\mathsf{v}', \mathsf{anchor})\}| \geq f + 1$ **then**
    ORDERANCHORS(anchor)

procedure ORDERANCHORS(v)
  anchor ← v
  orderedAnchorsStack.*push*(anchor)
  r ← anchor.*round* $- 2$
  while r > lastOrderedRound **do**
    prevAnchor ← GETANCHOR(r)
    if PATH(anchor, prevAnchor) **then**
      orderedAnchorsStack.*push*(prevAnchor)
      anchor ← prevAnchor
    r ← r − 2
  lastOrderedRound ← v.*round*
  ORDERHISTORY()

procedure ORDERHISTORY()
  while ¬orderedAnchorsStack.*isEmpty*() **do**
    anchor ← orderedAnchorsStack.*pop*()
    verticesToOrder ← $\{\mathsf{v} \in \bigcup_{\mathsf{r}>0} \mathsf{DAG_i}[\mathsf{r}] \mid \text{PATH}(\mathsf{anchor}, \mathsf{v}) \wedge \mathsf{v} \notin \mathsf{orderedVertices}\}$
    for every v ∈ verticesToOrder in some deterministic order **do**
      order v
      orderedVertices ← orderedVertices ∪ {v}

2.3 DAG construction

One naive way to disseminate the vertices and satisfy the DAG properties we need for ordering is to use reliable broadcast [2]. However, standard implementations require $O(n^2)$ communication per vertex. We use the Narwhal mempool system, which leverages a proof of availability mechanism, for an efficient and scalable implementation. The proof of availability mechanism allows Narwhal to separate data dissemination from the DAG construction, both of which is implemented with $O(n)$ communication complexity in the common case. In brief, parties keep steaming data to each other. A proof of a availability is a metadata (hash and a quorum of signatures) guaranteeing any party is able to retrieve the data. As a result, instead of blocks of transactions, each vertex of the DAG contains a list of proofs. After ordering the DAG, any party can retrieve the data in case it does not have it.

Advancing rounds. Each vertex in the DAG points to at least $n - f$ vertices in the previous round. Therefore, before advancing to round r and broadcasting a new vertex, a party must wait for $n - f$ vertices in round $r - 1$. Since there are at least $n - f$ honest parties, this can be done completely asynchronously allowing the DAG to grow in network speed. Indeed, this is exactly how asynchronous solutions like Aleph [5], Tusk [3] and DAG-Rider [6] advance rounds.

However, as we know from the famous FLP [4] result, deterministic partially synchronous consensus protocols has to use timeouts to satisfy liveness. Specifically, the problem in advancing rounds whenever $n - f$ vertices are delivered is that parties might not vote for the anchor even if the party that broadcast it is just slightly slower than the fastest $n - f$ parties. To deal with this, Bullshark integrates timeouts into the DAG construction. If the first $n - f$ vertices a party p gets in an even-numbered round r do not include the anchor of round r, then p sets a timer and waits for the anchor until the timer expires. Similarly, in an odd-numbered round, parties wait for either $f + 1$ vertices that vote for the anchor, or $2f + 1$ vertices that do not, or a timeout.

Responsiveness. It is important to note that the DAG construction satisfies responsiveness. That is, after GST, if the leader (the party that broadcasts the anchor) is honest, then the DAG is constructed in network speed and timeouts never expire. Moreover, since the Narwhal system disseminates data in network speed regardless the DAG construction, slightly slowing down the DAG construction does not effect the overall throughput (each vertex will simple have slightly more metadata).

Virtual DAG. One alternative we consider is to separate a physical DAG from a virtual DAG. In brief, the idea is that the physical DAG keeps advancing in network speed, i.e., parties advance physical rounds immediately after $n - f$ vertices are delivered in the current round. The logical DAG is piggybacked on top of the physical DAG - each vertex has an additional bit that indicates whether the vertex belongs to the logical DAG. A logical edge between two logical vertices exists if there is a physical path between them. The timeouts are integrated into the logical DAG similarly to how they are integrated into the DAG in the description above. The consensus logic runs on top of the logical DAG and once an anchor is committed, its entire causal history on the physical DAG is ordered.

One may expect that the logical DAG approach would increase the system throughput because parties never wait for a timeout to broadcast vertices on the physical DAG. However, as we explained above, since Narwhal separates data dissemination from the DAG construction, this makes no difference in practice. On the other hand, the latency in the logical DAG approach may actually increase – this is because if a logical vertex is not ready to be piggybacked when the physical vertex is broadcast, it will need to wait for the next physical round. Moreover, from the experience we gained implementing DAG-based protocols, we learned that slightly back pressuring (slowing down) the DAG construction leads to maximum performance since the vertices contain more metadata, which amortizes the overhead of building the DAG (e.g., networking, computation, memory, storage, etc). In fact, during the development we evaluated and compared both approaches and chose the inject timeouts in the physical layer as it significantly outperformed the decoupled solution (logical DAG) in both throughput and latency.

For more details please refer to the blogpost [1].

References

[1] Dag meets bft. `https://decentralizedthoughts.github.io/2022-06-28-DAG-meets-BFT/`, 2022.

[2] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[3] 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.

[4] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[5] Adam Gkagol, Damian Leśniak, Damian Straszak, and Michał Świketek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.

[6] 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*, pages 165–175, 2021.

[7] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[8] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical (full paper). *arXiv preprint arXiv:2201.05677*, 2022.