Beluga: Block Synchronization for BFT Consensus Protocols

Tasos Kichidis[*], Lefteris Kokoris-Kogias[*], Arun Koshy[*], Ilya Sergey[‡][*],
Alberto Sonnino[*][†], Mingwei Tian[*], Jianting Zhang[§],

[*]*Mysten Labs*
[†]*University College London*
[‡]*National University of Singapore*
[§]*Purdue University*

Abstract—**Modern high-throughput BFT consensus protocols use streamlined push-pull mechanisms to disseminate blocks and keep happy-path performance optimal. Yet state-of-the-art designs lack a principled and efficient way to exchange blocks, which leaves them open to targeted attacks and performance collapse under network asynchrony. This work introduces the *block synchronizer*, a simple abstraction that drives incremental block retrieval and enforces resource-aware exchange. Its interface and role sit cleanly inside a modern BFT consensus stack. We also uncover a new attack, where an adversary steers honest validators into redundant, uncoordinated pulls that exhaust bandwidth and stall progress. Beluga is a modular and scarcity-aware instantiation of the block synchronizer. It achieves optimal common-case latency while bounding the cost of recovery under faults and adversarial behavior. We integrate Beluga into Mysticeti, the consensus core of the Sui blockchain, and show on a geo-distributed AWS deployment that Beluga sustains optimal performance in the optimistic path and, under attack, delivers up to $3\times$ higher throughput and $25\times$ lower latency than prior designs. The Sui blockchain adopted Beluga in production.**

1. Introduction

The last decade of research in high-throughput Byzantine fault-tolerant (BFT) consensus protocols has unveiled that achieving world-class performance requires two key design choices. Firstly, modern blockchains decouple block ordering from bulk data dissemination [1], [2], and secondly, they chain the disseminated data with causal references to past disseminated data [2]–[4]. Both properties depart from the blueprint used by legacy protocols such as PBFT [5], where data is disseminated solely by a rotating leader using monotonically increasing view numbers; instead, they adopt a concurrent data dissemination approach. This means each validator is expected to assemble transactions into blocks or batches and broadcast them to all other validators [2].

This dissemination usually imposes chaining or causal dependencies between blocks, but protocols implement this with varying degrees of rigidity. Some protocols, such as Narwhal-Hotstuff [2] and Autobahn [6] achieve this through best-effort direct dissemination, after which a separate module imposes a causal order over this data. Others, such as HashGraph [7] and Blocklace-based systems [8], [9] have validators assemble transactions into a block and disseminate the block by referencing as many blocks as possible. Finally, protocols such as Bullshark [10] and Mysticeti [11] force validators to only disseminate blocks upon collecting at least a quorum of blocks from the previous round. Regardless of the approach, to implement atomic broadcast [12], the data dissemination layer must implement the reliable broadcast abstraction [12]. Specifically, even if the BFT consensus protocol ensures consistency, the dissemination layer must still enforce totality [12]: if one honest validator receives a block, then all other validators must be able to obtain it as well.

Through manual inspection of numerous high-performant BFT codebases [13]–[21], we observed that contrary to the liveness proofs and descriptions of these protocols, none of the state-of-the-art protocols actually implement an upfront reliable broadcast, such as Bracha broadcast [22]. That is, they do not implement the totality property through a double-echo. The reason is clear: this would be prohibitively expensive as it consumes precious bandwidth, and the double-echo is rarely needed (only upon faults or poor network conditions) [2]. We empirically uncover the *same* implicit two-phase pattern: an optimistic *push* of block identifiers followed by a probabilistic *pull* for missing blocks. We confirm this through a systematic inspection of diverse production and prototype codebases of state-of-the-art protocols, examining dissemination modules, recovery loops, timers, and traces. The optimistic push is implemented through weaker primitives such as Byzantine consistent broadcast (which does not guarantee totality) [14], [23], [24] or even best-effort broadcast [13], [16]. A recovery mechanism asynchronously *pulls* any missing blocks deemed "useful".

The missing component. Despite its importance, this push-pull mechanism remains unstructured glue code, often involving uncoordinated pre-dissemination of block identifiers, timer-driven random pulls, and no explicit bounds on recovery message complexity (Sec. 2.2). Protocols run ad hoc logic on received blocks to determine whether and when to trigger the pull part of the protocol. This hidden component governs throughput under load, tail latency, and resilience against adversarial behavior and network conditions. Yet it lacks a specification or provable bounds.

As an almost expected consequence of this discrepancy between theory and practice, we uncover how an adversary can induce a targeted performance degradation that we call the *pull induction attack*. Byzantine replicas selectively withhold messages during initial dissemination so that only a small subset of honest validators receive a block. This then triggers redundant and overlapping pull requests from the remaining honest validators. The attack can be repeated every round and can drastically increase latency and consume recovery bandwidth (Sec. 3). We even observe that some implementations entirely omit the pull mechanism, thus claiming happy path performance while silently failing to achieve liveness under adversarial settings.

In this work, we formalize the module of BFT consensus responsible for implementing the reliable broadcast abstraction, which we call the *block synchronizer* (Sec. 2.1). This module decides how validators push their blocks to other validators, how to fetch any missing causal dependencies, and runs a principled admission control on whether to include the block in its local dependency graph (i.e., use it as a parent). An ideal block synchronizer module must satisfy three goals: (G1) eventual availability, ensuring that every block can be retrieved; (G2) optimal push latency along the optimistic path; and (G3) bounded amplification even under adversarial scheduling. While existing block synchronizer protocols naturally achieve G1, they fall short on G2 due to conservative multi-step push procedures (such as explicit consistent broadcast) and on G3 because of ad hoc push-pull designs. We provide a summary of these protocols in Sec. 2.2 (Table 1).

We then present Beluga, a block synchronizer module that satisfies all the goals above (Sec. 4). Beluga is modular and integrates into existing protocols without altering their safety guarantees or ordering logic. The key insight behind the construction of Beluga is that heavyweight upfront dissemination proofs (such as running an explicit Bracha broadcast) are *sufficient but not necessary* for robustness. Instead, it structures the push phase to carefully select the causal history of blocks based on scarcity signals (as opposed to simply selecting the first that arrives) and opportunistically leverages implicit dissemination evidence. The detection of missing blocks is performed through the reception of messages containing digests of past blocks unknown to the validator. In other words, the pull mechanism leverages the chaining and causal history of messages. Together, these mechanisms enable scarcity-aware pulls that provably bound recovery complexity while retaining optimal optimistic-path performance. In short, Beluga aims to preserve the optimistic high performance of current implementations while reintroducing the robustness guarantees of true reliable broadcast.

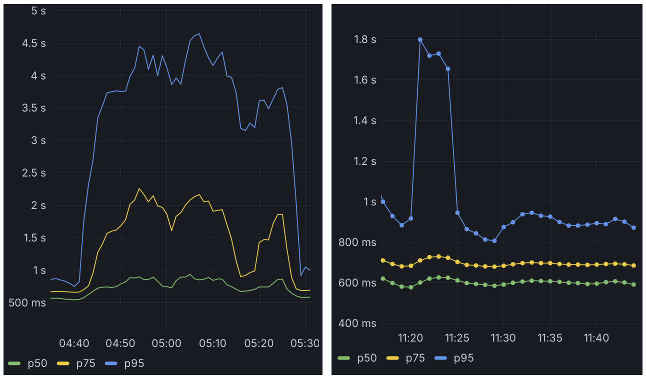


Figure 1. Latency comparison of the (production) Sui blockchain under network attacks before (left) and after (right) deploying Beluga.

Real world impact. We implement Beluga within Mysticeti, the consensus core of the Sui blockchain [25]. Our evaluation on a geo-distributed AWS testbed shows that Beluga maintains optimal optimistic-path latency and throughput while significantly improving performance under targeted attacks and adverse network conditions. Under attack scenarios, Beluga improves throughput by over 3× and reduces latency by 25× compared to baseline Mysticeti with its default block synchronizer (Sec. 6). We collaborated with the Sui team for over a year to integrate Beluga into the Sui blockchain. Beluga has been live since January 2025 with `mainnet-v1.42.0`, helping to secure over 10B USD in assets. Figure 1 shows the direct impact of Beluga: under network degradation attacks, Sui's tail latency improved by 5x after deploying Beluga.

Contributions. We make the following contributions:

1) We formalize the block synchronizer interface, threat model, and latency metrics, and derive baseline limitations of existing ad hoc mechanisms.
2) We design and analyze the *pull induction attack*, which increases latency by up to 50× and degrade throughput by up to 15× in existing systems.
3) We propose Beluga, a structured synchronizer with scarcity-aware detection and diversity-maximizing pull scheduling that preserves optimistic latency while bounding amplification.
4) We implement and integrate Beluga into Mysticeti/Sui, demonstrating no optimistic-path regression and significant under-attack throughput improvements (3×) and latency reductions (25×).

2. Problem Definition

Network model: We assume a set $\mathcal{V}$ of n validators (or parties; both are used interchangeably throughout this work) $\{v_1, \ldots, v_n\}$ and a static adversary $\mathcal{A}$ that can corrupt up to $f < n/3$ of the parties arbitrarily, at any point. A party is *crashed* if it halts prematurely at some point during execution. Parties that deviate arbitrarily from the protocol are called *Byzantine* or bad. Parties that are never crashed or Byzantine are called *honest*. Parties are communicating over a partially synchronous network [26], in which there exists a special event called Global Stabilization Time (GST) and a known finite time bound Δ (we use δ to represent the actual network latency), such that any message sent by a party at time x is guaranteed to arrive by time $\Delta + \max\{GST, x\}$.

Threat model The adversary is computationally bounded. Pairwise points of communication between any two honest parties are considered *reliable*, i.e., any honest message is *eventually* (after a finite, bounded number of steps) delivered. However, until GST the adversary controls the delivery of all messages in the network, with the only limitation that the messages must be eventually delivered. After GST, the network becomes synchronous, and messages are guaranteed to be delivered within Δ time after the time they are sent, potentially in an adversarially chosen order.

2.1. The Block Synchronizer Problem

We formalize the block synchronization problem here. We call it *block synchronizer*. As preliminaries, a block B contains the following basic data fields: round number r, block digest d, block creator $author$, connected parent blocks (in digest) $parents$, a list of transactions $payload$, and the creator's $signature$ on B.

Definition 1 (Block synchronizer)**.** *In the block synchronizer problem, a group of n validators $\mathcal{V}$ (of which up to f are Byzantine) collectively builds a structured, non-empty, and ever-growing set of blocks that are indexed by a monotonically increasing round number. Each validator v_i can call $block\_propose_i(B, r)$ to push its block B in round r (where $B.r{=}r$) to the system. Each validator v_i outputs $block\_accept_i(B.d)$ to accept B. The block B must include k blocks (from distinct validators) that $B.author$ has accepted in round $B.r{-}1$ as $B.parents$, where k is a parameter specifying how blocks are structured. Each validator v_i then outputs $block\_store_i(B)$ to store B. The protocol should satisfy the following properties:*

- ***Round-Progression:*** In each global round r of the system, at least $2f{+}1$ validators (not necessarily honest) invoke $block\_propose$ to create and disseminate their blocks. In other words, for each given $r \geq 0$, at least $2f{+}1$ produce blocks that have r as their round.
- ***Round-Termination:*** For each global round r of the system, every honest validator eventually outputs $block\_accept$ for the blocks produced in this round by least $2f{+}1$ validators. That is, for each given $r \geq 0$, each honest validator accepts block proposals, whose assigned round is r, from at least $2f{+}1$ different validators.
- ***Block availability***: If an honest validator v_i outputs $block\_accept_i(B.d)$ for some block B produced r, then v_i eventually outputs $block\_store_i(B)$.
- ***Causal availability***: If an honest validator v_i outputs $block\_accept_i(B.d)$ for some block B produced r, then for every block $B' \in causal(B)$, v_i eventually outputs $block\_accept_i(B'.d)$, where $causal(B)$ represents B's causal history (i.e., all blocks for which there is a connection or path from B to them).

2.2. Existing Synchronizer Protocols

Existing consensus protocols have some integrated block synchronization modules that are ad-hoc designed to guarantee progress. In this section, we explore these designs, each of which is composed of a push protocol and an *optional* pull protocol. The comparison is summarized in Table 1. Here, we model each class to the best of our abilities by both studying their research and implementations.

2.2.1. Multi-chain Certified Synchronizer Protocol.

Many multi-chain BFT protocols, such as Autobahn [6] and Star [27], are built on the Multi-chain certified synchronizer protocol. In the Multi-chain certified synchronizer protocol, validators structure blocks into multiple parallel chains, where each block includes only one block from the same validator in the last round as parents (i.e., $k{=}1$). The Multi-chain certified synchronizer protocol consists of a weak quorum-based push protocol to push blocks and a deterministic pull protocol to fetch missing blocks.

Weak quorum-based push protocol. For each round r, every validator v_i calls $block\_propose_i(B, r)$ to push a round r block B into the system. Specifically, upon outputting $block\_accept$ for its own block in round $r{-}1$, v_i moves to round r and create a new block B with v_i's round $r{-}1$ block in $B.parents$. Then, v_i employs a two-step Propose-Vote scheme to disseminate B. In the first step, v_i broadcasts B. In the second step, other validators respond to v_i with signatures on B and cache B. Note that validators do not output $block\_store$ at this point, since validators store a block only after it is certified. After that, v_i uses $f{+}1$ signatures to construct a *weak* certificate $WC(B.d)$ for B and outputs $block\_accept_i(B.d)$ and $block\_store_i(B)$. $WC(B.d)$ will be piggybacked into v_i's round $r{+}1$ block B'. The other validator v_j receiving B' will output $block\_accept_j(B.d)$ and output $block\_store_j(B)$ if receiving B.

Deterministic pull protocol. Validators in the Multi-chain certified synchronizer protocol perform the pull protocol independently from the push protocol. Specifically, when a validator v_i needs to pull a missing block B, v_i deterministically chooses a set of $f{+}1$ validators $\mathcal{V}_B$ in $WC(B.d)$ (i.e., validators who sign B) and sends a request message to $\mathcal{V}_B$. Since validators in $\mathcal{V}_B$ cache B, at least one honest validator can serve as the provider of B. Therefore, v_i must be able to receive B. v_i outputs $block\_store_i(B)$ once $block\_accept_i(B.d)$ succeeds and it receives B.

The Multi-chain certified synchronizer protocol ensures eventual availability (G1) and bounded amplification (G3), but not optimal optimistic-path push latency (G2). Specifically, it achieves a push latency (or round latency, both of them are used interchangeably throughout the paper) of 2δ with the Propose-Vote scheme and a bounded pull latency of 2δ (i.e., the time from when a node sends a pull request to when it receives the missing data) with the deterministic pull protocol. The pull complexity (i.e., the communication complexity per node of fetching a missing block) is $O(n)$.

TABLE 1. Comparison of synchronizer protocols and their integrations to BFT consensus, after GST and the leader is honest

Consensus Protocols	Synchronizer Protocols	Optimistic Case			Adverse Case[3]		
		Push Latency[1]	Consensus Latency[2]	Pull Complexity	Push Latency	Consensus Latency	Pull Complexity
Star [27]	Multi-chain certified	2δ	5δ	$O(n)$	2δ	5δ	$O(n)$
	Beluga	δ	5δ	$O(1)$	$\sim 2\delta$	$\sim 7\delta$	$O(n)$
Autobahn [6]	Multi-chain certified	2δ	7δ	$O(n)$	2δ	7δ	$O(n)$
	Beluga	δ	7δ	$O(1)$	$\sim 2\delta$	$\sim 9\delta$	$O(n)$
Bullshark [10]	DAG-based certified	3δ	6δ	$O(n)$	5δ	10δ	$O(n)$
	Beluga	δ	4δ	$O(1)$	$\sim 2\delta$	$\sim 8\delta$	$O(n)$
Shoal++ [28]	DAG-based certified	3δ	4δ	$O(n)$	5δ	8δ	$O(n)$
	Beluga	δ	3δ	$O(1)$	$\sim 2\delta$	$\sim 6\delta$	$O(n)$
Sailfish [29]	DAG-based RBC	4δ	5δ	None	4δ	5δ	None
	Beluga	δ	3δ	$O(1)$	$\sim 2\delta$	$\sim 6\delta$	$O(n)$
Sailfish++ [30]	DAG-based RBC	2δ	3δ	None	3δ	4δ	None
	Beluga	δ	3δ	$O(1)$	$\sim 2\delta$	$\sim 6\delta$	$O(n)$
Mysticeti [11]	DAG-based uncertified	δ	3δ	$O(1)$	$\geq 3\delta$	$\geq 9\delta$	$O(1)$
	Beluga	δ	3δ	$O(1)$	$\sim 2\delta$[4]	$\sim 6\delta$	$O(n)$

[1] Push latency is the network latency to create and disseminate a round of blocks. It measures the protocol's data dissemination speed. We consider the push latency is *optimal* if it equals δ throughout this paper.

[2] Consensus latency specifies how long the protocol requires to reach a consensus on a leader block.

[3] The adverse case means that the adversary performs the pull-induction attacks (detailed in Sec. 3.1), or for Sailfish++ [30], the optimistic termination condition is not satisfied.

[4] Beluga can achieve the push latency of nearly 2δ under the adverse cases (see Sec. 5.2 for more details).

2.2.2. DAG-based Certified Synchronizer Protocol. The certified DAG protocols, such as Bullshark [10], Shoal [31], and Shoal++ [28], are built on a DAG-based certified synchronizer protocol. In the DAG-based certified synchronizer protocol, validators organize blocks into a directed acyclic graph (DAG) using the quorum-based broadcast primitive. It consists of a consistent broadcast (CBC)-based push protocol to push certified blocks and a deterministic pull protocol to fetch missing blocks.

CBC-based push protocol. For each round r, every validator v_i calls $block\_propose_i(B, r)$ to push a round r block B into the system. Specifically, upon outputting $block\_accept$ for blocks in round $r{-}1$ from at least $2f{+}1$ validators, a validator v_i moves to round r and creates a new block B with *all* these round $r{-}1$ blocks as $B.parents$ (thus, $k{=}2f+1$ in DAG-based synchronizer protocols). Then, v_i employs a three-step certificate scheme to disseminate B. In the first step, v_i broadcasts B. In the second step, other validators respond to v_i with signatures on B and cache B. Note that validators do not output $block\_store$ at this point, since validators can store a block in the DAG only after it is certified. Then, v_i uses these $2f{+}1$ signatures to construct a certificate $C(B.d)$ for B. In the third step, v_i broadcasts $C(B.d)$. In addition, when receiving $C(B'.d)$, v_i outputs $block\_accept_i(B'.d)$ and output $block\_store_i(B')$ if receving B'.

Deterministic pull protocol. The DAG-based certified synchronizer protocol employs a deterministic pull protocol as used by the Multi-chain certified synchronizer protocol. A validator v_i will request a missing block from the set of validators $\mathcal{V}_B$ in $C(B.d)$. v_i outputs $block\_store_i(B)$ when $block\_accept_i(B.d)$ succeeds and it receives B.

The DAG-based certified synchronizer protocol ensures eventual availability (G1) and bounded amplification (G3), but not optimal optimistic-path push latency (G2). Specifically, it can achieve a push latency of 3δ with the CBC-based push protocol and a pull latency of 2δ with the deterministic pull protocol. The pull complexity is $O(n)$.

2.2.3. DAG-based RBC Synchronizer Protocol. Many recent DAG-based BFT consensus protocols, such as Sailfish [29] and its variant Sailfish++ [30], are built on a DAG-based RBC synchronizer protocol. In the DAG-based RBC synchronizer protocol, validators organize blocks into a DAG using the reliable broadcast (RBC) primitive. The RBC ensures that if an honest validator receives a block, all honest validators will eventually receive it (i.e., ensuring the totality property). Therefore, the pull protocol is *theoretically* not needed in the DAG-based RBC synchronizer protocol, and it only contains an RBC-based push protocol.[1]

[1] However, we find the implementations of the DAG-based RBC synchronizer protocol remain built on Bullshark's codebase, which still employs a pull mechanism [24], [32], likely due to the practical challenges of implementing upfront RBC (discussed in Sec. 1).

RBC-based push protocol. For each round r, every validator v_i calls $block\_propose_i(B, r)$ to push a round r block B into the system. Specifically, upon outputting $block\_accept$ for blocks in round $r{-}1$ from at least $2f{+}1$ validators, a validator v_i moves to round r and creates a new block B referencing *all* these round $r{-}1$ blocks as $B.parents$. v_i disseminates B using an RBC protocol [22], [30], [33], [34]. If v_i is honest, according to the RBC's Validity property, every other honest validator v_j will eventually deliver B (i.e., v_j will receive B). In addition, when v_i delivers B' that is reliably broadcast by another validator v_j, v_i checks if it can accept B' by checking if it has outputted $block\_accept_i(B''.d)$ for every block $B'' \in casual(B')$. If yes, v_i outputs $block\_accept_i(B'.d)$ and $block\_store_i(B')$ right after. If no, v_i will put B' into a pending list and update the list whenever it outputs $block\_accept$ for a new block.

The DAG-based RBC ensures eventual availability (G1) but fails to guarantee optimal optimistic-path push latency (G2) and bounded amplification (G3). Specifically, implementing an upfront reliable broadcast requires honest validators to continually transmit messages to an unresponsive adversary, leading to unbounded retransmissions. In addition, the push latency depends on the RBC protocol used, but none of them are optimal (i.e., δ). For instance, Sailfish [29] employs the RBC protocol from Das et al. [34], leading to a push latency of 4δ. Sailfish++ [30] adopts the state-of-the-art RBC protocol to achieve a push latency of 2δ (still not optimal) in optimistic cases (where at least $\lceil \frac{n+2f-2}{2} \rceil$ validators behave honestly). Sailfish++'s RBC protocol has a round latency of 3δ when optimistic cases do not hold.

2.2.4. DAG-based Uncertified Synchronizer Protocol.

The Mysticeti [11] consensus protocol uses a DAG-based uncertified synchronizer protocol. Validators structure blocks into a DAG, where each block includes at least $2f{+}1$ blocks from the last round as parents. The DAG-based uncertified synchronizer protocol consists of a best-effort broadcast (BEB)-based push protocol to push blocks and a random pull protocol to fetch missing blocks.

BEB-based push protocol. For each round r, every validator v_i calls $block\_propose_i(B, r)$ to push a round r block B into the system. When receiving a block B' from other validators, v_i checks if it has outputted $block\_accept_i(B''.d)$ for every block $B'' \in casual(B')$. If yes, v_i outputs $block\_accept_i(B'.d)$ and $block\_store_i(B')$ right after. If no, v_i uses the pull protocol to get *all* missing blocks. In essence, v_i must synchronize the whole causal history of B' before outputting $block\_accept$. Upon outputting $block\_accept$ for blocks in round $r{-}1$ from at least $2f{+}1$ validators, v_i moves to round r and creates a new block B with *all* these round $r{-}1$ blocks as $B.parents$. v_i broadcasts B in a best-effort way.

Random pull protocol. To pull a missing block B in the DAG-based uncertified synchronizer protocol, the validator v_i randomly chooses a constant set of validators $\mathcal{V}_B \subseteq \mathcal{V}$ and sends a request message to $\mathcal{V}_B$. v_i repeatedly sends the request message to different sets of validators at intervals until receiving B. However, since B is not certified and v_i requests B from randomly selected validators, there is no guarantee that v_i will ever receive B. After getting B, v_i repeats the pull protocol to get all missing blocks of $causal(B)$. During the pull process, v_i outputs $block\_accept_i(B.d)$ and $block\_store_i(B)$ consecutively once it has synchronized B and all blocks in $causal(B)$.

The DAG-based uncertified protocol achieves eventual availability (G1) and optimal optimistic-path push latency (G2), but not bounded amplification (G3). Specifically, it achieves the round latency of δ with best-effort broadcast under optimistic cases. However, due to its random pull, it introduces uncertain pull latency and unbounded pull requests, thereby potentially unbounded push latency for every future round, despite having $O(1)$ pull complexity.

3. Pull Induction Attacks and Key Insights

Despite the rich design space proposed by prior work, none of them have taken a principled approach and achieved all ideal goals (G1-G3, Sec. 1). This allows us to exploit their vulnerabilities through a new class of attacks we call pull induction attacks, which deliberately trigger unnecessary pulls to degrade their performance. This section sheds light on the pain points of existing protocols and provides several key insights that guide the design of Beluga.

3.1. Pull Induction Attacks

The goal of pull induction attacks is to induce honest validators to pull blocks from others, thereby increasing the round latency. To this end, the adversary selectively disseminates its blocks to only a subset of honest validators. Consequently, validators that do not receive the adversary's blocks are compelled to pull the missing blocks that are included by the protocol. Table 1 (adverse case column) presents a performance comparison of different block synchronizer protocols under pull induction attacks.

We give a concrete example of a pull induction attack against the DAG-based uncertified synchronizer protocol adopted by Mysticeti. We select Mysticeti as an example system since it is a state-of-the-art protocol widely adopted in industry [25], [35]. We leave the discussion of pull induction attacks against other synchronizer protocols in the full paper. In Figure 2, there are four validators v_1, v_2, v_3, and v_4, of which v_4 is the adversary. In round $r{-}1$, v_4 only disseminates its round $r{-}1$ block B_4^{r-1} to v_1, making v_1's round r block B_1^r reference B_4^{r-1}. When pushing round r blocks and receiving B_1^r, both v_2 and v_3 miss B_4^{r-1}, and thus, they invoke the random pull protocol to fetch B_4^{r-1} before accepting B_1^r and having enough (i.e., 3 with $n{=}4$ and $f{=}1$) accepted round r blocks to propose their round $r{+}1$ blocks. The latency of round r thus consists of δ for pushing round r blocks and at least 2δ for pulling B_4^{r-1}. Similarly, in round r, v_4 only disseminates its round r block B_4^r to v_2, making v_2's round $r{+}1$ block B_2^{r+1} reference B_4^r. This will induce both v_1 and v_3 to pull B_4^r before accepting

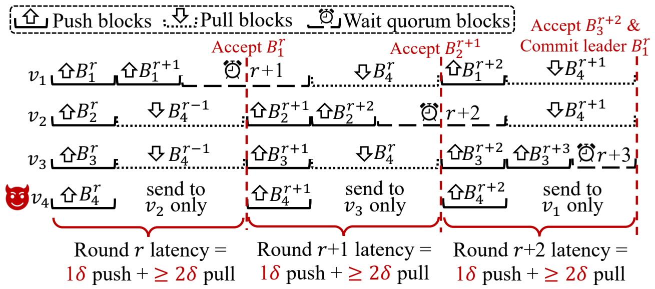


Figure 2. The pull induction attacks: the adversary v_4 selectively shares its blocks with honest validators. In round $r-1$, v_4 only shares its round $r-1$ block B_4^{r-1} with v_1, making v_1's round r block B_1^r reference B_4^{r-1}. This will induce v_2 and v_3 to trigger the pull protocol to fetch B_4^{r-1} to accept B_1^r, eventually increasing the latency of round r. Similarly, v_4 only shares its round r block with v_2 and shares its round $r+1$ block with v_3. This will increase the latency of each round by at least one pull round-trip.

B_2^{r+1}, thereby increasing the latency of round $r+1$ to at least 3δ. In round $r+1$, v_4 only disseminates its round $r+1$ block B_4^{r+1} to v_3, which will induce both v_1 and v_2 to pull B_4^{r+1} before accepting B_3^{r+2}, thereby increasing the latency of round $r+2$ to at least 3δ. Recall that the consensus of Mysticeti requires three rounds of blocks [11, Algorithm 3]. As a result, the consensus latency of Mysticeti is at least 9δ under this pull induction attack.

3.2. Key Insights

After reviewing the existing block synchronizer protocols and their behavior under pull induction attacks, we have the following insights.

Key insight 1: DAG-based synchronizer protocols can accelerate the consensus latency, but at risk of being delayed by the adversary. In DAG-based synchronizer protocols, each block references at least $2f+1$ parents, and the connections between blocks can serve as proposal votes. This allows validators to complete the consensus during the formation of the DAG without extra communication by interpreting the DAG structure locally. As a result, the consensus latency of the protocols built on DAG-based synchronizer is soundly low under happy cases (e.g., 3δ in Mysticeti and 4δ in Sailfish). In contrast, the consensus protocols built on Multi-chain certified must rely on a dependent consensus process to order blocks, which introduces extra communication overhead. For instance, apart from 2δ round latency for pushing a round of blocks, Autobahn requires an additional 5δ network latency to reach a consensus on a leader block, leading to the consensus latency of 7δ.

However, forcing a block to reference $2f+1$ parents enables the adversary to delay the progress of the synchronizer protocol with the pull induction attacks. Specifically, when an honest validator has its block reference some adversary blocks that are not shared with other honest validators, these honest validators must pull the missing adversary blocks before accepting the honest validator's block and collecting enough $2f+1$ accepted blocks to move to the next round. This increases the round latency by at least one pull round-trip. As a result, both round latency and consensus latency increase under the pull induction attacks.

Key insight 2: Block certificates allow performing the push and pull protocols separately. In the Multi-chain certified synchronizer protocol, validators push each block accompanied by a certificate containing a quorum of $\geq f+1$ signatures, thereby attesting the block and causal availability. In this case, validators can output *block_accept* for each certified block they receive without pulling any missing blocks in the block's causal history while still ensuring the Causal availability property. This enables the pull protocol to be performed in separation from the push protocol. We call this feature pulling blocks *off the push path*, as pulling blocks can be completed independently from the push protocol. In contrast, blocks are pulled *on the push path* if pulling missing blocks is necessary before validators accept them.

Supporting pulling blocks off the push path prevents Byzantine validators from proactively delaying the progress of the synchronizer protocol via pull induction attacks, since pulling missing blocks does not prevent validators from creating new blocks. However, the Multi-chain certified synchronizer protocols adopt an explicit certificate mechanism, where certificates are created within at least one round-trip latency, leading to higher push and consensus latencies compared to the DAG-based uncertified synchronizer protocol.

Key insight 3: The pull protocol introduces a trade-off between round latency and communication complexity. In certified synchronizer protocols (e.g., DAG-based certified and Multi-chain certified), the pull is deterministic, where validators pull missing blocks from a specific set of validators (with the set size $O(n)$). This ensures validators can fetch the missing blocks within a constant round trip (i.e., 2δ with one for sending requests and the other for receiving blocks). However, this also introduces high communication complexity per validator (i.e., $O(n)$) since each validator might receive redundant blocks from others.

In contrast, the DAG-based uncertified synchronizer protocol adopts a random pull protocol, where validators randomly pull the missing blocks from a small set of validators (with the set size $O(1)$). This reduces the communication complexity per validator to $O(1)$. However, such a random pull cannot ensure a constant round trip for synchronizing missing blocks. This is not a problem when the protocol is under happy cases, as pulling blocks does not impede progress. However, under adverse cases, the DAG-based uncertified synchronizer protocol requires at least 2δ in the pull protocol, leading to at least 3δ round latency and at least 9δ consensus latency.

4. The Beluga Protocol

4.1. Overview

Beluga is an efficient and robust DAG-based block synchronizer protocol composed of two key components: an AC-based optimistic push protocol (Sec. 4.2) and a hybrid pull protocol (Sec. 4.3) based on the novel idea of Implicit Proof-of-Availability (ImPoA).

AC-based optimistic push. Motivated by Insight 1, Beluga adopts a DAG structure and employs an optimistic push protocol, similar to the DAG-based uncertified synchronizer, to achieve optimal round latency under happy cases (G2) and bounded retransmission (G3). Specifically, validators disseminate blocks using a best-effort broadcast, resulting in δ round latency. However, unlike the DAG-based synchronizer protocols, where validators arbitrarily reference parent blocks, Beluga introduces an *admission control (AC)* mechanism to filter out blocks based on the creators' behaviors. With AC, honest validators avoid referencing parent blocks created by suspected Byzantine validators—specifically, those that previously triggered them to invoke the pull protocol. This mechanism effectively safeguards Beluga against pull induction attacks.

ImPoA-based hybrid pull. Motivated by Insights 2 and 3, Beluga's pull protocol aims to separate pulling from pushing and balance pull complexity and latency. To this end, Beluga introduces an Implicit Proof-of-Availability (ImPoA) mechanism, which enables validators to identify blocks that can be safely accepted even if some of their ancestors are temporarily unavailable, thereby enabling validators to pull blocks off the push path. Based on ImPoA, we categorize pulling blocks into two types: *live blocks* and *bulk blocks*. Live blocks contribute to the quorum formation of the latest round but have unavailable ancestors; validators must accept these blocks to progress to the next round and propose new ones. In contrast, bulk blocks do not contribute to quorum formation in the current round. Leveraging this distinction, Beluga employs a hybrid pull strategy: live blocks are pulled deterministically to minimize the latency, while bulk blocks are pulled randomly to reduce pull complexity. The pull protocol allows Beluga to achieve eventual availability (G1) and bounded push latency (G3).

Block structure. To capture validators' behaviors during the block pushing process and facilitate the pull process, Beluga augments the block structure with three additional fields.

- *Weak links*: The $weaklinks$ field references blocks that a validator has received and accepted but not selected as parents. We therefore called $parents$ as strong links, both are used interchangeably throughout the paper.
- *Watermark*: The $watermark$ is an n-element array maintained by each validator, where the i-th entry implies the highest round number of block received from validator v_i.
- *Ancestors*: The $ancestors$ is an n-element array storing, for each validator v_i, the highest round number of v_i's blocks reachable from the current block.

4.2. AC-based Optimistic Push Protocol

Beluga's push protocol specifies how validators create blocks and disseminate their created blocks to others. Block dissemination in Beluga is optimistic and relies on a best-effort broadcast; that is, validators disseminate blocks to all others without waiting for acknowledgments. Block creation is governed by an *Admission Control* (AC) module, which enforces rules that filter blocks according to the creators' behaviors as quantified by a *reputation* mechanism. Figure 8 (deferred to Appendix B due to page limitation) illustrates Beluga's AC-based optimistic push protocol. We detail the reputation mechanism and the AC module below.

Reputation Mechanism (Lines 23-32, Figure 8). Each validator v_i maintains a local reputation table TR_i that records the reputations of all validators. The reputation entry $TR_i[j]$ reflects the contribution of validator v_j to the block pushing process. Specifically,

- *Reputation Increase:* v_i increases v_j's reputation if it receives $2f+1$ blocks (denoted by $\mathcal{B}_r$) in round r that collectively indicate v_j's round $r-1$ block has been received. Formally, if for each block $B \in \mathcal{B}_r$, $B.watermark[j] = r-1$, then $TR_i[j]$ is incremented.
- *Reputation Decrease:* v_i decreases v_j's reputation whenever (i) it invokes the pull protocol to fetch a missing block created by v_j, or (ii) it receives pull requests for a v_j's block from at least $f+1$ distinct validators. A pull request serves as a *report* of v_j's delayed dissemination behavior. Collecting $f+1$ such reports constitutes a *blame* against v_j, indicating that at least one honest validator failed to receive v_j's block in a timely manner. Thus, v_i will decrease v_j's reputation with a blame.

Intuition behind the reputation mechanism. The reputation mechanism is designed to capture validators' behaviors during the block pushing process. Specifically, if a validator v_i consistently pushes its blocks to all other validators, honest validators will frequently observe $2f+1$ blocks indicating that v_i's latest block is disseminated and received timely, thereby increasing v_i's reputation. In contrast, if v_i selectively pushes its blocks to only a subset of validators—performing a pull induction attack that forces others to invoke the pull protocol to retrieve its blocks—its reputation will decrease. Consequently, honest validators naturally maintain high reputations, whereas malicious validators that frequently launch such attacks accumulate low reputations.

In Beluga, the reputation increase is set by 1, while the reputation decrease is set by a large value R_L (e.g., R_L=10,000). This asymmetry prevents the adversary from launching unbounded pull induction attacks and eventually enables Beluga to achieve a decent round and consensus latencies. A detailed analysis is presented in Sec. 5.2.

Admission Control (Lines 1-22, Figure 8). The AC module determines which blocks are selected as parents for newly created blocks based on their creators' reputations. When creating a new block in round r, a validator v_i first collects the latest blocks it has received from all validators with round numbers $\leq r-1$, denoted by the set $\mathcal{B}^{r-1}$. It then selects parent blocks from $\mathcal{B}^{r-1}$ through the following steps: (i) filter out any block in $\mathcal{B}^{r-1}$ that is not from round $r-1$ or is deemed unacceptable; (ii) from the remaining set, select the top $2f+1$ blocks in based on their creators' reputations in TR_i. A block is considered *acceptable* if v_i has received all of its ancestors or can otherwise ensure its ancestors are available (see Sec. 4.3.1 for more details). This AC mechanism ensures that honest validators avoid referencing blocks created by suspected Byzantine validators—specifically, those with low reputations due to previously triggering pull requests. By excluding such malicious blocks, an honest validator can create new blocks whose ancestors have already been received and accepted by all other honest validators, without invoking the pull protocol during the push process. Consequently, Beluga is inherently protected against pull induction attacks.

Apart from parents, a validator also references other acceptable blocks in $\mathcal{B}^{r-1}$ as $weaklinks$ in the new block. The $weaklinks$ serve as evidence of block availability to facilitate the pull process. We discuss it in Sec. 4.3.

4.3. ImPoA-based Hybrid Pull Protocol

Beluga's pull protocol defines how validators fetch missing blocks when they cannot be accepted during the push process. Beluga's pull protocol is called the implicit proof-of-availability (ImPoA)-based hybrid pull protocol. As illustrated in Figure 3, it comprises two components: an ImPoA-based pull mechanism and a hybrid pull strategy.

4.3.1. ImPoA-based Pull Mechanism. The ImPoA-based pull mechanism enables validators to pull certain blocks off the push path once their availability can be proven. However, unlike the existing certified synchronizer protocols (such as Multi-chain certified) that create *explicit* block certificates at the cost of additional communication overhead during the push process, Beluga constructs *implicit* block certificates by locally interpreting block patterns.

Implicit PoA. In Beluga, a block B is identified as *implicitly available* if it is referenced (via strong or weak links) by at least $f{+}1$ blocks from the subsequent rounds. Note that a validator references B only if it (i) receives B, and (ii) can verify the availability of B's causal history (lines 4-5, Figure 8). Consequently, these $f{+}1$ referencing blocks collectively form an implicit proof-of-availability (PoA) for B, implying that at least one honest validator can attest to B's causal availability. For instance, in Figure 3(a), the missing block B_1^{r+1} is identified implicitly available as it was referenced by two received blocks B_2^{r+2} and B_4^{r+2}.

Live and Bulk modules. With ImPoA, validators can safely accept blocks even when parts of their causal histories are missing, thereby allowing missing blocks to be pulled off the push path. To accommodate the mechanism, Beluga employs two pull synchronizer modules: (i) *bulk synchronizer module*, which retrieves missing blocks off the push path, and (ii) *live synchronizer module*, which retrieves missing blocks on the push path.

Mechanism specification. We now describe the workflow of the ImPoA-based pull mechanism. Assume a validator v_i is currently proceeding in round r. Upon receiving a block B containing missing parents (i.e., B has strong links to some blocks that v_i has not accepted) during the push process, v_i determines which pull synchronizer module will be used to pull B's missing ancestors based on B's round number. We denote $\mathcal{B}_{bk}$ the block set in the bulk synchronizer module and $\mathcal{B}_{lv}$ the block set in the live synchronizer module.

If $B.r{<}r$, it means that v_i receives an old-round block that will not be included in v_i's threshold clock (i.e., block quorum for round r). The pull synchronization of B will not affect the push protocol, and thus B could be synchronized off the push path. v_i then transmits B to the bulk synchronizer module and includes B into $\mathcal{B}_{bk}$.

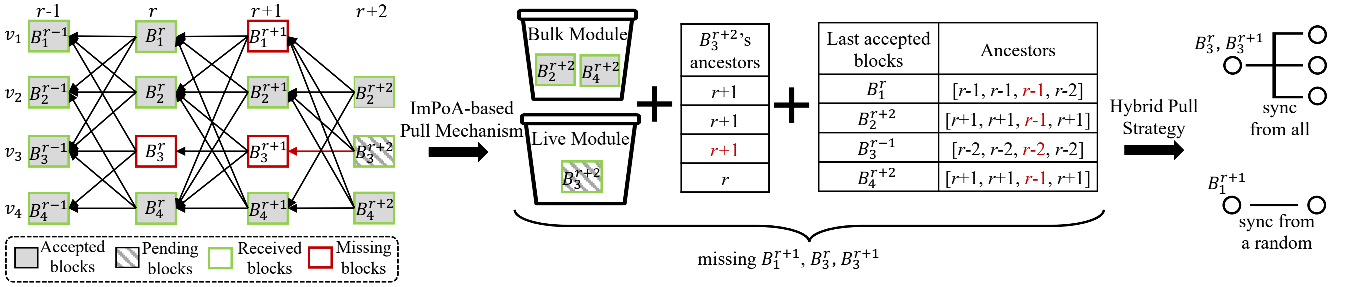
If $B.r{\geq}r$, it means that B is a live block from v_i's local view. However, instead of transmitting B to the live synchronizer module immediately, v_i first checks whether B can be proven available. Specifically, if every parent of B has an implicit PoA from the live block set $\mathcal{B}_{lv}$ or has been $block\_accept$, then v_i ensures B is available. v_i then transmits B to the bulk synchronizer, outputs $block\_accept$ and $block\_store$ for B, and include B into the bulk block set $\mathcal{B}_{bk}$. If, otherwise, B is not proven available; v_i then transmits B to the live synchronizer module and includes B into $\mathcal{B}_{lv}$. For example, in Figure 3(a), v_4 is proceeding in round $r{+}2$ and transmits B_2^{r+2} and B_4^{r+2} to the bulk synchronizer module even though their common parent B_1^{r+1} was missing, since B_1^{r+1} has an implicit PoA. In contrast, B_3^{r+2} is transmitted to the live synchronizer module.

Moreover, v_i can dynamically transmit blocks from $\mathcal{B}_{lv}$ to $\mathcal{B}_{bk}$ according to the following conditions:

1) Upon advancing to a new round $r' > r$, v_i transmits any $B' \in \mathcal{B}_{lv}$ to $\mathcal{B}_{bk}$ if $B'.r < r'$.
2) Upon including a new block into $\mathcal{B}_{lv}$, v_i checks whether any block $B' \in \mathcal{B}_{lv}$ is proven available. If yes, v_i transmits such available B' from $\mathcal{B}_{lv}$ to $\mathcal{B}_{bk}$.

Identifying missing blocks. By utilizing the ImPoA-based pull mechanism, Beluga classifies blocks with missing ancestors into two categories: *Live blocks* (i.e., blocks in $\mathcal{B}_{lv}$) and *Bulk blocks* (i.e., blocks in $\mathcal{B}_{bk}$). Each validator v_i can then locally identify missing blocks from $\mathcal{B}_{lv}$ and $\mathcal{B}_{bk}$ by checking their $ancestors$. However, since blocks in the DAG are well-connected, $\mathcal{B}_{lv}$ and $\mathcal{B}_{bk}$ might involve overlapped missing ancestors. For instance, in Figure 3, the missing block B_1^{r+1} is the common ancestor of both the live blocks and the bulk blocks. As a result, v_i might pull the same missing blocks in both the live and bulk synchronizer modules redundantly.

To avoid redundant synchronization, v_i traces the last accepted blocks from all validators and combines them with blocks' ancestors to identify missing blocks for distinct synchronizer modules. Since each block must reference the previous block proposed by the same validator [11], the last accepted block B_j^r from v_j in round r indicates that all blocks from v_j with a round $r' \leq r$ are proven available and can be accepted by v_i. Moreover, if a validator accepts a block, it means that the block's causal history is available and can be accepted. As a result, with the information, v_i can identify the missing blocks that are *only* required by the live blocks.


r-1
r
r+1
r+2
v_1 B_1^{r-1} B_1^{r} B_1^{r+1}
v_2 B_2^{r-1} B_2^{r} B_2^{r+1} B_2^{r+2}
v_3 B_3^{r-1} B_3^{r} B_3^{r+1} B_3^{r+2}
v_4 B_4^{r-1} B_4^{r} B_4^{r+1} B_4^{r+2}
Accepted blocks
Pending blocks
Received blocks
Missing blocks
ImPoA-based Pull Mechanism
Bulk Module
B_2^{r+2} B_4^{r+2}
Live Module
B_3^{r+2}
missing B_1^{r+1}, B_3^r, B_3^{r+1}
Hybrid Pull Strategy
B_3^r, B_3^{r+1}
sync from all
B_1^{r+1}
sync from a random

B_3^{r+2}'s ancestors
r+1
r+1
r+1
r

Last accepted blocks	Ancestors
B_1^r	[r-1, r-1, r-1, r-2]
B_2^{r+2}	[r+1, r+1, r-1, r+1]
B_3^{r-1}	[r-2, r-2, r-2, r-2]
B_4^{r+2}	[r+1, r+1, r-1, r+1]

(a) The local view of the DAG

(b) Classify bulk and live blocks, and identify missing blocks

(c) Send pull requests

Figure 3. The ImPoA-based hybrid pull protocol for v_4: (a) v_4 is proceeding in round r+2 but misses blocks B_1^{r+1}, B_3^r, and B_3^{r+1}. B_1^{r+1} is identified as implicitly available as $\{B_2^{r+2}, B_4^{r+2}\}$ reference it. As a result, v_4 accepts B_2^{r+2} and B_4^{r+2} even though their parent B_1^{r+1} is not received. (b) v_4 transmits blocks that reference missing ones to the bulk and live synchronizer modules via the ImPoA-based pull mechanism. With the hints, v_4 identifies the missing blocks it needs to fetch. (c) v_4 fetches blocks via a hybrid pull strategy, balancing pull latency and complexity.

As shown in Figure 3(b), the last accepted blocks and their $ancestors$ information imply that v_4 (i) will be able to accept r+1 block B_1^{r+1} from v_1, and (ii) has accepted round r+2 block B_2^{r+2} from v_2, round $r-1$ block B_3^{r-1} from v_3, and round r+2 block B_4^{r+2} from itself. When processing the pending block B_3^{r+2} (which references v_3's round r+1 block) in the live synchronizer module, v_4 can identify B_3^r and B_3^{r+1} are only required by the live blocks (i.e., B_3^{r+2}) but B_1^{r+1} does not. As a result, the live synchronizer module only requests the missing blocks B_3^r and B_3^{r+1}, while the bulk synchronizer module requests the missing block B_1^{r+1}.

4.3.2. Hybrid Pull Strategy. After identifying the missing blocks that a validator v_i needs to fetch, v_i deploys a hybrid pull strategy to balance message and round complexities, as shown in Figure 3(c).

Pulling blocks in the live synchronizer. Blocks $\mathcal{B}_{lv}$ in the live synchronizer module are time-sensitive, as their missing blocks can block the push process. Thus, the live synchronizer module adopts a *deterministic pull strategy* to minimize the latency. Specifically, v_i sends the pull requests for all missing blocks specified in $\mathcal{B}_{lv}$ to all validators. Such a pull strategy might pull redundant blocks. However, it guarantees that v_i can receive missing blocks as long as they are $block\_accepted$ by one honest validator, and process live blocks within a round-trip delay (i.e., 2δ).

Pulling blocks in the bulk synchronizer. Pulling blocks in the bulk synchronizer does not block the push process, and thus, is not sensitive to latency. Consequently, the bulk synchronizer module adopts a *random pull strategy* to minimize the pull complexity. Specifically, for each missing block specified in $\mathcal{B}_{bk}$, v_i randomly chooses one validator to send the pull request. If v_i does not receive the requested missing block within some predefined time (say Δ_{bk}), it randomly chooses another validator to send the pull request.

After getting missing blocks, v_i outputs $block\_accept$ and $block\_store$ for each of them if v_i hasn't done it before.

4.4. Building BFT Consensus on Beluga

Beluga can be integrated into any existing BFT consensus protocols as an independent module. Figure 4 illustrates how Beluga interfaces with a generic BFT state machine replication system. In short, Beluga implements an efficient and robust RBC abstraction. It provides the consensus module with accepted blocks for ordering and guarantees the availability of the ordered blocks for execution. The key enabler of the generality is that the DAG structure in Beluga implies the *RBC patterns* on blocks, allowing different BFT consensus protocols to apply their own consensus rules over these patterns to achieve consistent block ordering.

RBC pattern. In Beluga, validators can determine whether a block has been reliably broadcast by interpreting the DAG locally. Specifically, if a block is referenced by at least $2f$+1 subsequent blocks, it forms an RBC pattern, indicating that the block has been reliably broadcast (our pull protocol ensures totality). A block exhibiting such a pattern is called an *RBC block*. Generating an RBC block B_R involves two push rounds in Beluga. Any block that references this RBC pattern, i.e., includes the $2f$+1 blocks that reference B_R, is considered to be voting for B_R.

Applying consensus rules on Beluga. With the RBC patterns, a BFT consensus protocol can apply its underlying consensus rules on top of Beluga. Table 1 (Sec. 2) summarizes the performance comparison between the original BFT consensus and its variant incorporating Beluga.

For multi-chain protocols such as Dashing [27] and Autobahn [6], validators execute separate consensus instances. A leader validator uses its most recent RBC block (and blocks implying the RBC pattern, which certifies the causal availability) as a proposal to coordinate a consensus instance. This design achieves improved push latency in optimistic conditions, at the cost of a modest increase in consensus latency under adverse conditions.

For DAG-based protocols, the consensus is performed by interpreting the DAG locally. Beluga enhances these protocols by reducing both push and consensus latencies while improving robustness. In these protocols, certain blocks are designated as leader blocks to drive consensus progress.

During the push phase, Beluga's admission control will include leader blocks and those voting for the leader blocks, from validators with benign reputations. To perform ordering, validators check whether the leader RBC blocks satisfy the consensus rules defined by the BFT consensus protocol itself. Distinct DAG-based BFT consensus protocols may employ different consensus rules or impose additional constraints (e.g., Sailfish [29] enforces validators to either vote for leader blocks or conduct no-vote certificates). Nevertheless, they can share the same interfaces provided by Beluga.

Specifically, for Bullshark [10] and its successor Shoal [31], their consensus operates in two rounds of RBC blocks: a leader RBC block is directly committed once at least $f+1$ subsequent RBC blocks vote for it. When executing their consensus rules atop Beluga, validators can order blocks within four push rounds (with two rounds of RBC blocks), achieving a consensus latency of 4δ in optimistic cases and 8δ in adverse cases.

For Shoal++ [28], Sailfish [29], and Sailfish++ [30], their consensus operates in one round of RBC blocks plus the first messages of another round of RBC blocks. Specifically, a leader RBC block is directly committed once at least $2f+1$ first messages of the next-round RBC blocks vote for it. Since the first message of an RBC block corresponds to a single block in Beluga, integrating Beluga allows them to order blocks within three push rounds, achieving a consensus latency of 3δ in optimistic cases and 6δ in adverse cases.

Finally, Mysticeti [11] and Cordial Miners [36], when equipped with Beluga, achieve the same optimal push and consensus latencies in optimistic scenarios, while offering greater robustness under adverse conditions. This improvement arises because Beluga constructs a DAG using a similar best-effort broadcast approach (but with distinct admission control and pull mechanisms).

Garbage collection. Beluga naturally reuses the garbage collection mechanism introduced by Narwhal [2]; this component allows the protocol to clean up any partially disseminated messages that were not promptly committed, preventing unbounded storage and memory growth. Notably, this module is employed in most implementations, both in DAG-based and linear BFT protocols [13], [14], [19].

5. Analysis

5.1. Correctness Analysis

In this section, we prove Beluga satisfies the properties defined in Definition 1 (Sec. 2.1).

Lemma 1. *After GST, all honest validators will enter the same round within* 3Δ.

Proof. W.l.o.g, assume round r is the last round entered by honest validators before GST. After GST, the message delay between honest validators is bounded by Δ. Thus, all honest validators must receive at least one round r block B_i^r from honest validator v_i by time $GST+\Delta$. Every honest validator can synchronize all B_i^r's parent blocks and missing ancestors via the pull protocol within 2Δ. Consequently, all honest validator can accept at least $2f+1$ round $r-1$ blocks and enter round r by time $GST+3\Delta$. □

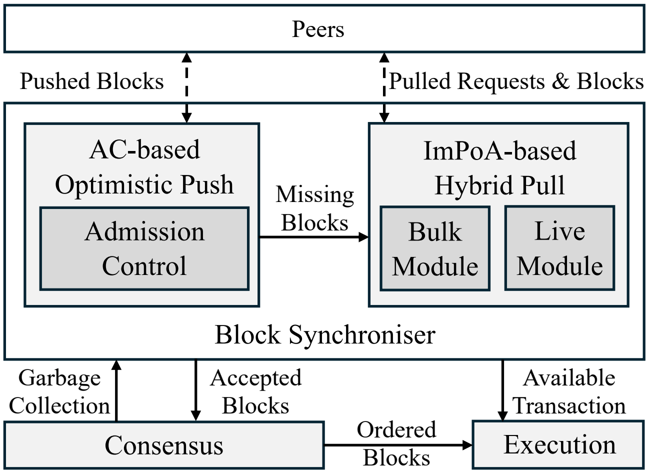


Figure 4. Beluga can be integrated into any BFT consensus protocol. By applying the consensus rules on the blocks produced by Beluga, validators derive a consistent order for blocks. Beluga guarantees the availability of ordered blocks for the state machine replication (SMR) execution.

Lemma 2. *After GST, if an honest validator v_i enters round r at time t_r, and all honest validators have created and disseminated their round r blocks by time t_r, then all honest validators will be able to enter round $r+1$ by time $t_r+3\Delta$.*

Proof. According to this lemma's condition, t_r is the time when the slowest honest validator creates and disseminates its round r block. By time $t_r+\Delta$, all honest validators will have received the round r blocks from all honest validators. Even though some honest validators might need to synchronize missing ancestors to accept some round r blocks, they can accept these blocks via our pull protocol within the time bound of 2Δ. As a result, all honest validators can accept at least $2f+1$ round r blocks by time $t_r+3\Delta$ and enter their round $r+1$ blocks by time $t_r+3\Delta$. □

Theorem 1 (Block availability)**.** *Beluga satisfies block availability. If an honest validator v_i outputs $block\_accept_i(B.d)$ for some block B produced r, then v_i eventually outputs $block\_store_i(B)$*

Proof. In Beluga, an honest validator v_i outputs $block\_accept_i(B.d)$ for some block B in round r when v_i received B. As a result, v_i must have stored B and output $block\_store_i(B)$. □

Theorem 2 (Causal availability)**.** *Beluga satisfies causal availability. If an honest validator v_i outputs $block\_accept_i(B.d)$ for some block B, then for every block $B' \in causal(B)$, v_i eventually outputs $block\_accept_i(B'.d)$, where $causal(B)$ represents B's causal history.*

Proof. In Beluga, an honest validator v_i outputs $block\_accept_i(B.d)$ for some block B in round r when v_i received B and ensures all its parents in round $r-1$ are available—that is, v_i has either outputted $block\_accept$ for the parent blocks or observed they are referenced by at least $f{+}1$ subsequent blocks (Sec. 4.3). In the latter case where the parent blocks (denoted by a set $\mathcal{B}^{r-1}$) are referenced by at least $f{+}1$ subsequent blocks, according to Beluga's push protocol, the creators of these $f{+}1$ subsequent blocks must have outputted $block\_accept$ for $\mathcal{B}^{r-1}$. This means that at least one honest validator v_j has stored $\mathcal{B}^{r-1}$ and ensured all parents of $\mathcal{B}^{r-1}$ in round $r{-}2$ are available. As a result, v_j must have either outputted $block\_accept$ for the parent blocks of $\mathcal{B}^{r-1}$ in round $r{-}2$ or observed they are referenced by at least $f{+}1$ subsequent blocks. By induction, we can see that for every block $B' \in causal(B)$, there exists at least one honest validator that has stored B' and ensured all its parents are available. As a result, v_i can eventually receive B' from this honest validator via the pull protocol and output $block\_accept_i(B'.d)$. □

Theorem 3 (Round-Progression)**.** *Beluga satisfies round-progression. For each round $r \geq 0$, at least $2f{+}1$ validators will create and disseminate their round r blocks.*

Proof. For the genesis round 0, all validators will create and disseminate their round 0 blocks. Thus, the lemma holds for round 0. In addition, according to Lemma 1, all honest validators can enter the same round (w.l.o.g. at round r) within 3Δ after GST. As a result, all (i.e., at least $2f{+}1$) honest validators must be able to create and disseminate their round r blocks by time $GST{+}3\Delta$. Thanks to Beluga's pull protocol, every honest validator can accept at least $2f{+}1$ round r blocks and create its round $r{+}1$ block. By induction, we can see that for any future round $r'{>}r$, at least $2f{+}1$ validators will create and disseminate their round r' blocks. Moreover, for any round $1 \leq r'' \leq r$, since Beluga's push protocol requires validators to reference at least $2f{+}1$ parent blocks from the previous round when creating their round r'' blocks, at least $2f{+}1$ validators must have created and disseminated their round $r'' - 1$ blocks. By induction, we can see that for any previous round $1 \leq r'' \leq r$, at least $2f{+}1$ validators must have created and disseminated their round r'' blocks. The proof is done. □

Theorem 4 (Round-Termination)**.** *Beluga satisfies round-termination. For each round $r \geq 0$, each honest validator accepts block proposals, whose assigned round r, from at least $2f{+}1$ different validators.*

Proof. For the genesis round 0, since all honest validators create their round 0 blocks, and round 0 blocks do not reference any blocks, each honest validator can accept at least $2f{+}1$ round 0 blocks. The lemma holds for round 0. In addition, according to Theorem 3, for each round $r \geq 1$, at least $2f{+}1$ validators will create and disseminate their round r blocks. Each round r blocks consist of at least $f{+}1$ blocks created by honest validators. For these $f{+}1$ honest validators, according to Beluga's push protocol, they must output $block\_accept$ to accept at least $2f{+}1$ round $r{-}1$ blocks. According to Theorem 1 and Theorem 2, these round $r{-}1$ blocks and their causal histories are available to all honest validators. As a result, each honest validator can accept at least $2f{+}1$ round $r{-}1$ blocks. By induction, we can see that for any round $r \geq 1$, each honest validator can accept at least $2f{+}1$ round r blocks. The proof is done. □

5.2. Performance Analysis

In this section, we will show that Beluga can achieve optimal round latency (i.e., Δ) under happy cases and the round latency of nearly 2Δ under adverse cases after GST.

5.2.1. Round latency under happy cases. In happy cases, all responsive validators (at least $2f{+}1$) are honest and share their blocks in time. According to Lemma 1, all honest validators can enter the same round (w.l.o.g. at round r) within 3Δ after GST and can create their round r blocks by time $GST{+}3\Delta$. As a result, every honest validator must be able to receive a quorum $\mathcal{B}^r$ containing at least $2f{+}1$ round r blocks by time $GST{+}4\Delta$. Since all responsive validators are honest in happy cases, every honest validator will receive $\mathcal{B}^r$ at time, w.l.o.g., $t_{sync}{<}GST{+}4\Delta$ and move to the next round $r{+}1$ at t_{sync}. After that, all honest validators can receive at least $2f{+}1$ round $r{+}1$ blocks by time $t_{sync}{+}\Delta$ and move to round $r{+}2$ at time $t_{sync}{+}\Delta$. By induction, we can see that the round latency under happy cases is Δ.

5.2.2. Round latency under adverse cases. We give a sketch proof for Theorem 5 below and defer the detailed proof to Appendix A.

Theorem 5. *Beluga can achieve a round latency of nearly 2Δ under adverse cases.*

Proof (sketch). To avoid being blamed and penalized in reputation, the adversary must share its blocks with at least $f{+}1$ honest validators within a time interval of Δ. Specifically, if the adversary v_m shares its block B_m with an honest validator v_i at time t_i and with another honest validator v_j at time $t_j{>}t_i$, then $t_j{-}t_i{<}\Delta$, since otherwise, v_j will need to fetch B_m after receiving v_i's block referencing B_m, thereby reporting v_m. For the remaining honest validators that do not receive B_m directly from v_m, they can either utilize Beluga's ImPoA pull mechanism to avoid fetching B_m during their pushing phase, or propose two consecutive-round blocks within 4Δ (cf. Lemma 6).

On the other hand, if the adversary shares its block with fewer than $f{+}1$ honest validators, it will be blamed and its reputation reduced by R_L each time. Once the adversary's reputation falls below that of honest validators, Beluga 's admission control will prevent its blocks from being referenced via strong links by honest validators, resulting in a round latency of Δ thereafter (Lemma 4). Consequently, for each R_L reputation loss, the adversary must behave appropriately (i.e., sharing its blocks with at least $f{+}1$ honest validators) for at least R_L rounds to restore its reputation. During the recovery phase, the system reverts to the above case, yielding a round latency of 2Δ. By setting R_L sufficiently large, we can effectively force the adversary to perform properly, leading to a round latency of nearly 2Δ. □

6. Experimental Evaluation

We evaluate the throughput and latency of Mysticeti [11] equipped with Beluga through experiments conducted on Amazon Web Services (AWS) on a geo-distributed testbed. Appendix C describes our implementation in detail. We then show its improvements over the baseline Mysticeti implemented with the default push-pull block synchronizer module (Sec. 2). We chose to implement Beluga within Mysticeti because (1) it is a state-of-the-art DAG-based BFT consensus protocol deployed in production in multiple blockchains [25], [35], providing real-world impact, (2) its open-source codebase [16] is well documented and modular, facilitating implementation, and (3) it builds upon an uncertified DAG, making it particularly vulnerable to the attacks described in Sec. 3.1. These properties make Mysticeti an ideal candidate to demonstrate the effectiveness of Beluga.

Our evaluation demonstrates the following claims:

C1 Beluga introduces no noticeable performance overhead when the protocol runs in ideal conditions (no Byzantine parties and synchronous network)

C2 Beluga drastically improves both latency and throughput in the presence of asynchronous network conditions and the attack presented in Sec. 3.1.

Note that evaluating the performance of BFT protocols in the presence of generic Byzantine faults is an open research question [37], and state-of-the-art evidence relies on formal proofs.

Appendix D describes our experimental setup in detail.

6.1. Benchmarks in Ideal Conditions

Figure 5 depicts the performance of Mysticeti equipped with both Beluga and the baseline block synchronizer operating through a best-effort push protocol followed by a random pull (described in Sec. 2) running with 50 honest validators. As expected, the performance of the baseline Mysticeti is similar to Mysticeti equipped with Beluga. Both systems exhibit a stable throughput up to around 100,000 tx/s while maintaining a latency of around 0.5s, and both scale easily to 300,000 tx/s (512-byte transactions) while maintaining sub-second latency (0.8s). We did not push throughput further for cost reasons, but both systems appear to use less than 10% of their CPU at that point, indicating that they could likely handle even higher loads. This performance similarity is due to Beluga's lightweight requirements during the happy path: when all parties are honest and the network is synchronous, Beluga imposes minimal constraints on parent block selection and thus operates similarly to the baseline push-pull synchronizer. This confirms Item **C1** that Beluga does not introduce any significant overhead when the network is synchronous and all parties are honest.

6.2. Benchmarks under Attack

Figure 6 depicts the performance of Mysticeti with both Beluga and the baseline block synchronizer in a 10-validator deployment with 1 or 3 faults (we limit this benchmark to 10 validators for cost reasons). The Byzantine validators perform the pull induction attack described in Sec. 3.1, creating conditions that can also arise from severe network asynchrony. The baseline Mysticeti suffers severe throughput degradation and dramatic latency increases. For reference, the graph includes the no-fault performance with 10 validators to illustrate the performance gap under attack. With three Byzantine faults, baseline Mysticeti's throughput drops by over 15x and its latency increases to over 50 seconds, compared to 0.5 seconds in the fault-free case. In contrast, Mysticeti equipped with Beluga maintains substantially higher throughput: Beluga rapidly detects and prioritizes inclusion of blocks from reliable validators in the DAG. The throughput reduction (30%) is primarily attributable to the loss of faulty validator capacity. Latency increases by approximately 4x to around 2 seconds, which is expected given the need to wait for additional blocks before proceeding to the next round and to recover from attack-induced asynchrony. This represents a substantial improvement over baseline Mysticeti: over 3x higher throughput and 25x lower latency with three Byzantine faults. These results confirm Item **C2** that Beluga significantly improves both latency and throughput under asynchronous network conditions and pull induction attacks.

7. From paper to Mainnet

The motivation for developing Beluga arose in March 2024 following a testnet incident on the Sui blockchain, triggered by a fundamental change in the protocol's networking stack. Some validators misconfigured their machines, resulting in a situation where they could receive blocks from other validators but were unable to propagate their own. Consequently, these validators accumulated a large number of blocks that remained unshared with the network. In one extreme case, a single validator locally created approximately 750,000 blocks that had not been disseminated. Once the validator corrected their configuration and broadcast all these blocks simultaneously, the network experienced a stall: fast validators began including these blocks in their DAGs, and once $f+1$ of them did so, the rest of the network had to synchronize as well, causing a substantial performance degradation. Specifically, all 750,000 blocks were committed within one minute, triggering an overwhelming number of pull requests.

This incident revealed the root cause in the block synchronizer component. Its admission control module (see Figure 4) lacked sufficient filtering for poorly performing validators, allowing the most recent hoarded blocks to enter local DAGs. Moreover, the original pull protocol exacerbated the problem by indiscriminately pulling from both poorly performing and random validators, resulting in high synchronization latency. These observations underscored the need for a high-quality DAG that minimizes synchronization from underperforming validators.

We collaborated with the Sui team to experiment with various heuristics inspired by leader scoring [38], but initially, excluding ancestors indiscriminately disrupted block

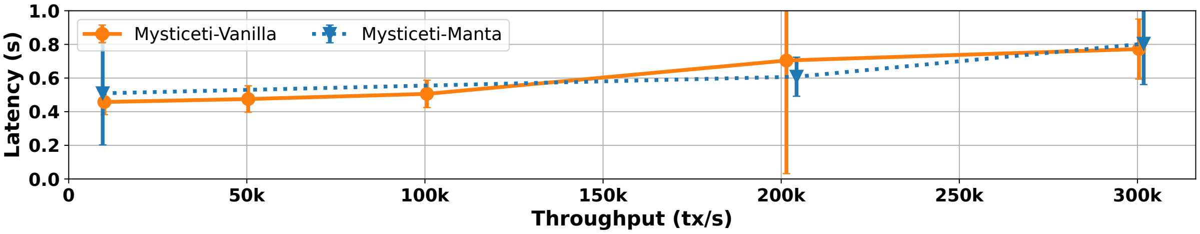


Figure 5. Comparative throughput-latency performance of Mysticeti equipped with Beluga and with the baseline push-pull block synchronizer. WAN measurements with 50 validators, no faults, and 512B transaction size.

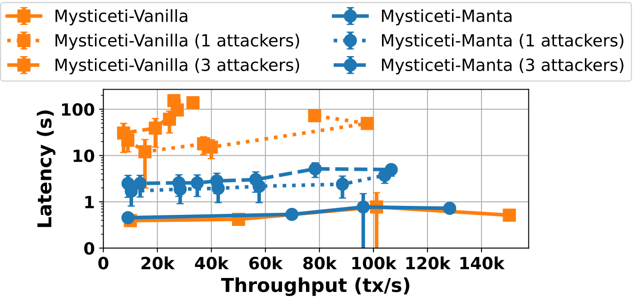


Figure 6. Comparative throughput-latency performance of Mysticeti equipped with Beluga and with the baseline push-pull block synchronizer. WAN measurements with 10 validators, zero, one, and three faults, and 512B transaction size. (Note the log scale on latency.)

propagation even under benign conditions. This motivated the introduction of optional dependencies (Sec. 4). Subsequently, they embarked on a multi-month effort to collect detailed consensus metrics and evaluate different strategies for scoring validators. This process ultimately led to the design presented in Section 4. After nearly a year of testing and tuning, Beluga was deployed on Sui mainnet version `mainnet-v1.42.0` in January 2025.

Figure 1 (Sec. 1) shows a production network replicating the Sui mainnet with all 135 validators, in their respective geo-location and with distribution stake, sustaining a constant load of 6,000 transactions per second. The figure compares the original Mysticeti protocol run within the Sui mainnet before Beluga deployment, and the improved version with Beluga. The results demonstrate that Beluga effectively mitigates the latency spikes previously observed in the network under attack, resulting in a stable and predictable transaction commit latency. Beluga effectively prevents poorly performing validators from adversely impacting the overall network performance, resulting in a 5x reduction in the 95th percentile, a 2x reduction in the 75th percentile, and a 25% reduction in the 50th percentile latency.

Figure 7 shows the rate at which blocks from an intentionally slow validator (green line) appear in the committed DAG under the same production setup after deploying Beluga. The validator's contribution to the committed DAG falls to zero within roughly 1-2 minutes as its score drops and the admission-control module at other validators discards its blocks. It remains zero throughout the attack, and then returns to its original rate within 3 minutes of recovering. Notice how other validators remain largely unaffected (other colored lined on the figure). This result demonstrates that Beluga rapidly identifies and deprioritizes poorly performing validators, preventing them from degrading network performance during misbehavior, and then promptly restores their ability to participate once they return to normal behavior. As a result, recovering validators rejoin the consensus process without long-term penalties.

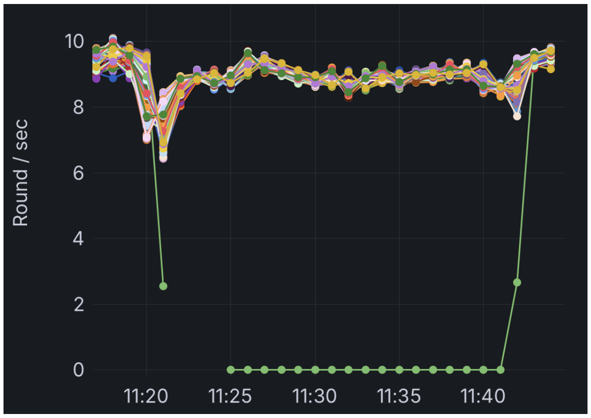


Figure 7. Block proposal rate of an intentionally slow validator in a Sui mainnet reproduction with 135 validators after deploying Beluga. Each colored line represents the proposal rate of a different validator. The validator's proposal rate quickly drops to zero during the attack and recovers shortly after performance is restored.

8. Related Work

A largely unexplored dimension of the consensus literature concerns the design of synchronization primitives themselves. While many protocols assume a specific synchrony model, prior work, to our knowledge, does not isolate, specify, or evaluate such primitives as first-class, modular components of consensus stacks. We address this gap by defining and implementing synchronization abstractions that compose cleanly with existing consensus protocols.

One key source of inspiration is the Pacemaker [39] of leader-based protocols. The pacemaker separates view-change timing, leader rotation, and timeout management from the safety-critical core, which creates clean interfaces and lets liveness mechanisms evolve independently. Our synchronizer plays a similar role: it fits into the modular blockchain architecture [40], defines isolated processes that deliver high-performance support to the safety-critical total-ordering algorithm, and lets engineers separate concerns.

A synchronizer protocol can be viewed as a decomposed form of reliable broadcast [41], one of the most studied problems in distributed computing. Unlike AVID [42], [43] protocols, which focus on communication complexity in asynchronous settings, or optimistic reliable broadcast [30], which emphasizes latency under a low number of faults, our design examines performance after GST in partially synchronous or synchronous networks. This approach provides a more practical perspective and has already enabled significant performance improvements in Sui, a top-20 blockchain.

Acknowledgments

This work is funded by Mysten Labs and was conducted while Jianting Zhang was interning with the company. We thank George Danezis, Adrian Perrig, and Philipp Jovanovic for their insightful discussions that greatly improved this work.

References

[1] L. Yang, V. Bagaria, G. Wang, M. Alizadeh, D. Tse, G. Fanti, and P. Viswanath, “Prism: Scaling bitcoin by 10,000 x,” *arXiv preprint arXiv:1909.11261*, 2019.

[2] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, “Narwhal and tusk: a dag-based mempool and efficient bft consensus,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[3] E. Buchman, “Tendermint: Byzantine fault tolerance in the age of blockchains,” Ph.D. dissertation, University of Guelph, 2016.

[4] S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, “Be aware of your leaders,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 279–295.

[5] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[6] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks, “Autobahn: Seamless high speed bft,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 1–23.

[7] O. Green, “Hashgraph—scalable hash tables using a sparse graph data structure,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 2, pp. 1–17, 2021.

[8] P. S. Almeida and E. Shapiro, “The blocklace: A byzantine-repelling and universal conflict-free replicated data type,” 2025. [Online]. Available: https://arxiv.org/abs/2402.08068

[9] E. Shapiro, “Grassroots systems: Concept, examples, implementation and applications,” 2024. [Online]. Available: https://arxiv.org/abs/2301.04391

[10] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, “Bullshark: Dag bft protocols made practical,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2705–2718.

[11] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, “Mysticeti: Reaching the latency limits with uncertified dags,” in *Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[12] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[13] MystenLabs, “Sui blockchain,” https://github.com/mystenlabs/sui, 2025, accessed: 2025.

[14] A. Sonnino, “Narwhal and tusk implementation,” https://github.com/asonnino/narwhal, 2025, accessed: 2025.

[15] ——, “Hotstuff implementation,” https://github.com/asonnino/hotstuff/tree/3-chain, 2025, accessed: 2025.

[16] M. Labs, “Mysticeti: Low-latency dag consensus with fast commit path,” https://github.com/asonnino/mysticeti, 2024.

[17] A. Sonnino, “Jolteon implementation,” https://github.com/asonnino/hotstuff, 2025, accessed: 2025.

[18] Diem Association, “Diem blockchain,” https://github.com/diem/diem, 2025, accessed: 2025.

[19] N. Giridharan, “Autobahn artifact,” https://github.com/neilgiri/autobahn-artifact, 2025, accessed: 2025.

[20] P. Tennage, “Mahi-mahi consensus implementation,” https://github.com/PasinduTennage/mahi-mahi-consensus, 2025, accessed: 2025.

[21] D. Xiang, “Ditto implementation,” https://github.com/danielxiangzl/Ditto, 2025, accessed: 2025.

[22] G. Bracha, “Asynchronous byzantine agreement protocols,” *Information and computation*, vol. 75, no. 2, pp. 130–143, 1987.

[23] A. Sonnino, “Bullshark implementation,” https://github.com/asonnino/narwhal/tree/bullshark, 2025, accessed: 2025.

[24] Nibesh Shrestha, “Sailfish codebase,” https://github.com/nibeshrestha/sailfish, 2025, accessed: 2025.

[25] “Sui,” https://sui.io/, 2024.

[26] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[27] S. Duan, H. Zhang, X. Sui, B. Huang, C. Mu, G. Di, and X. Wang, “Dashing and star: Byzantine fault tolerance with weak certificates,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 250–264.

[28] B. Arun, Z. Li, F. Suri-Payer, S. Das, and A. Spiegelman, “Shoal++: High throughput dag bft can be fast!” *arXiv preprint arXiv:2405.20488*, 2024.

[29] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak, “Sailfish: Towards improving the latency of dag-based bft,” *Cryptology ePrint Archive*, 2024.

[30] N. Shrestha, Q. Yu, A. Kate, G. Losa, K. Nayak, and X. Wang, “Optimistic, signature-free reliable broadcast and its applications,” in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[31] A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li, “Shoal: Improving DAG-BFT latency and robustness,” in *Financial Cryptography and Data Security (FC 2024), Revised Selected Papers, Part I*, ser. Lecture Notes in Computer Science, vol. 14744. Cham: Springer, 2025, pp. 92–109, fC 2024, Willemstad, Curaçao, March 4–8, 2024. [Online]. Available: https://doi.org/10.1007/978-3-031-78676-1_6

[32] Qianyu Yu, “Sailfish++ codebase,” https://github.com/qyu100/SFSailfish/tree/OptSFSailfish, 2025, accessed: 2025.

[33] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, “Good-case latency of byzantine broadcast: A complete categorization,” in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 331–341.

[34] S. Das, Z. Xiang, and L. Ren, “Asynchronous data dissemination and its applications,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2705–2721.

[35] T. I. team, https://docs.iota.org/about-iota/iota-architecture/consensus, 2025.

[36] I. Keidar, O. Naor, O. Poupko, and E. Shapiro, "Cordial miners: Fast and efficient consensus for every eventuality," 2023. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2023.26

[37] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," in *ACM PODC*, 2021.

[38] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, "Hammerhead: Leader reputation for dynamic scheduling," 2023. [Online]. Available: https://arxiv.org/abs/2309.12713

[39] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain," *The Libra Assn., Tech. Rep*, vol. 7, 2019.

[40] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Proof of availability and retrieval in a modular blockchain architecture," in *International Conference on Financial Cryptography and Data Security*. Springer, 2023, pp. 36–53.

[41] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, 1989.

[42] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 2005, pp. 191–201.

[43] N. Alhaddad, S. Das, S. Duan, L. Ren, M. Varia, Z. Xiang, and H. Zhang, "Asynchronous verifiable information dispersal with near-optimal communication," *Cryptology ePrint Archive*, 2022.

[44] T. T. Team, "Tokio," https://tokio.rs, 2024.

[45] H. de Valence, "Ed25519 for consensus-critical contexts," https://crates.io/crates/ed25519-consensus, 2024.

[46] RustCrypto, "Rustcrypto: Hashes," https://github.com/RustCrypto/hashes, 2024.

[47] Die.Net, "writev(3) - linux man page," https://linux.die.net/man/3/writev, 2024.

[48] Meta, "Sapling (minibytes)," https://github.com/facebook/sapling/tree/main/eden/scm/lib/minibytes, 2024.

[49] T. S. Team, "Validator deployment amd configuration," https://docs.sui.io/guides/operator/validator/validator-config, 2025.

Appendix A. Performance Analysis under Adverse cases

In this section, we give a rigorous proof to show that Beluga can achieve a round latency of nearly 2Δ under adverse cases. The proof relies on the following assumption. (Note that the correctness of Beluga (cf. Section 5.1) does not rely on these assumptions.)

Assumption 1 (Latency Triangle). *After GST, the direct network latency between any pair of honest validators is always faster than going through an intermediate validator.*

Assumption 1 is reasonable, since the direct communication path is usually the shortest path.

In adverse cases, there are at most f malicious validators that aim to increase the round latency by inducing honest validators to trigger the pull protocol. In the following, we denote the set of honest validators as $\mathcal{V}_h$ and the set of malicious validators as $\mathcal{A}$.

Lemma 3. *After GST, each honest validator will not get blamed and have its reputation decreased by honest validators.*

Proof. Recall in Beluga (Section 4.2), a validator v_i will get its reputation decreased by honest validators if $f+1$ validators report that they invoke the pull protocol to synchronize v_i's blocks. An honest validator v_j invokes the pull protocol when it receives a block B_k^r from another validator V_k that references v_i's round $r-1$ block B_i^{r-1}, but v_j has not received B_i^{r-1} yet. However, according to Assumption 1, after GST, if v_i is honest and sends its block B_i^{r-1} to v_j, then v_j must receive B_i^{r-1} directly before receiving it through an intermediate validator (i.e., through V_k creating B_k^r) indirectly. Therefore, v_j will not invoke the pull protocol to synchronize B_i^{r-1} and will not report v_i if v_i is honest. Since there are at most f malicious validators, an honest validator will not be reported by $f+1$ validators and will not get blamed by honest validators. The proof is done. □

Lemma 4. *After GST, if all honest validators enter round r at time t_r and have their reputation higher than that of any malicious validator, then for any future round $r' \geq r$, the latency of round r' is Δ.*

Proof. Since all honest validators have a higher reputation than any malicious validator, according to our reputation-based push protocol, honest validators will reference round $r-1$ blocks from honest validators only when creating their round r blocks. There is no need to invoke the pull protocol to accept these round r blocks after GST. As a result, the latency of round r is Δ, and all honest validators can enter round $r+1$ at time $t_r+\Delta$. By induction, we can see that the latency of any future round $r' \geq r$ is Δ. The proof is done. □

Lemma 5. *After GST, if all honest validators enter round r at time t_r, then either the expected latency of any future round $r'>r$ is within 2Δ or at least one malicious validator is blamed by honest validators, in which case the latency of round r' is at most 3Δ.*

Proof. Recall that if an honest validator v_i enters round r at time t_r, then v_i must have received at least $2f+1$ acceptable round $r-1$ blocks and can create its round r block at t_r. Since all honest validators enter round r at time t_r, every honest validator will receive at least $2f+1$ round r blocks created by honest validators by time $t_r+\Delta$. There are three cases.

Case 1: If all honest validators have their reputation higher than that of any malicious validator, then according to Lemma 4, the latency of future round $r'>r$ is Δ.

Case 2: If it is not the case 1, and for each round $r-1$ block created by malicious validators $\mathcal{A}$, it is referenced by at least $f+1$ round r blocks created by honest validators, i.e., $\mathcal{A}$ share their round $r-1$ blocks with at least $f+1$ honest validator. In this case, thanks to the ImPoA-based pull mechanism, every honest validator can accept all round r blocks created by honest validators without synchronizing any missing blocks on the push path and create its round $r{+}1$ block at time $t_r{+}\Delta$. The latency of round $r{+}1$ is Δ. By induction, we can see that for any future round $r' \geq r$, the latency of round r' is Δ for case 2.

Case 3: If it is not the case 1, and at least one round $r{-}1$ block B_m^{r-1} created by a malicious validator V_m is referenced by fewer than $f{+}1$ round r blocks created by honest validators, i.e., V_m delays or did not share B_m^{r-1} with more than f honest validators. We denote those honest validators referencing B_m^{r-1} by t_r as $\mathcal{V}_h^{Ref}$, and those who do not as $\mathcal{V}_h^{NoR}$. There are two scenarios, and we show that for any future round $r' \geq r{+}1$, the expected latency of r' is at most 2Δ before V_m is blamed.

First, V_m will never share B_m^{r-1} with $\mathcal{V}_h^{NoR}$, then $|\mathcal{V}_h^{NoR}| \geq f{+}1$ honest validators will report V_m, and V_m will be blamed by honest validators.

Second, V_m delays sharing B_m^{r-1} with some honest validators $\mathcal{V}_{h1}^{NoR}$ but not the others $\mathcal{V}_{h2}^{NoR}$ to escape being blamed, where $|\mathcal{V}_{h2}^{NoR} = \mathcal{V}_h^{NoR} \setminus \mathcal{V}_{h1}^{NoR}| \leq f$. In this scenario, note that $\mathcal{V}_{h1}^{NoR}$ must have received B_m^{r-1} by time $t_r{+}\Delta$, since otherwise, $\mathcal{V}_{h1}^{NoR}$ learn B_m^{r-1} is missing from $\mathcal{V}_h^{Ref}$'s round r blocks and will report V_m. Thus, both $\mathcal{V}_h^{Ref}$ and $\mathcal{V}_{h1}^{NoR}$ can create their round $r{+}1$ blocks by time $t_r{+}\Delta$. The delayed honest validators $\mathcal{V}_{h2}^{NoR}$, instead, must pull B_m^{r-1} at time $t_r{+}\Delta$ (when they receive round r blocks from $\mathcal{V}_h^{Ref} \cup \mathcal{V}_{h1}^{NoR}$) and pull any missing round r blocks at time $t_r{+}2\Delta$ (when they receive round $r{+}1$ blocks from $\mathcal{V}_h^{Ref} \cup \mathcal{V}_{h1}^{NoR}$). As a result, even for the delayed honest validators $\mathcal{V}_{h2}^{NoR}$, they can create their round $r{+}1$ blocks by time $t_r{+}3\Delta$ and create their round $r{+}2$ blocks $t_r{+}4\Delta$. Since the non-delayed $\mathcal{V}_h^{Ref} \cup \mathcal{V}_{h1}^{NoR}$ can create their round $r{+}2$ blocks by time $t_r{+}4\Delta$, such a delaying process can be repeated every two rounds. As a result, the maximum average latency of each future round $r' \geq r{+}1$ is delayed by at most 2Δ, if V_m wishes to escape being blamed.

In addition, when V_m is blamed due to it delaying a round r'', according to Lemma 2, the latency of round r'' is at most 3Δ. The proof is done.

□

Lemma 6. *After GST, once all honest validators enter round r, for future round $r'{>}r$ that malicious validators $\mathcal{A}$ delay, its expected latency is within 2Δ, or at least one malicious validator is blamed by honest validators.*

Proof. After GST, according to Lemma 1, all honest validators can enter the same round (w.l.o.g. at round r) within 3Δ. Let $t_r{=}GST{+}3\Delta$. Note that at least one honest validator V_{fst} must have created its round r block B_{fst}^r before GST, and all honest validators must receive B_{fst}^r by $GST{+}\Delta$.

Consider two sets of honest validators: slow honest validators $\mathcal{V}_h^{slw}$ and fast honest validators $\mathcal{V}_h^{fst} = \mathcal{V}_h \setminus \mathcal{V}_h^{slw}$. Validators in $\mathcal{V}_h^{slw}$ need to invoke the pull protocol for missing round $r{-}1$ blocks to enter round r by time t_r, while validators in $\mathcal{V}_h^{fst}$ do not. Apparently, we have $|\mathcal{V}_h^{fst}| \geq f{+}1$, since otherwise, there are more than $f{+}1$ honest validators in $\mathcal{V}_h^{slw}$ reporting $\mathcal{A}$, and $\mathcal{A}$ would be blamed. Moreover, note that all validators in $\mathcal{V}_h^{fst}$ must be able to create their round r blocks by time $GST{+}\Delta$ (i.e., right after receiving B_{fst}^r), since otherwise, they need to invoke the pull protocol to fetch missing round $r{-}1$ blocks referenced by B_{fst}^r, contradicting to the definition of $\mathcal{V}_h^{fst}$. As a result, all validators in $\mathcal{V}_h^{slw}$ must receive round r blocks from all validators in $\mathcal{V}_h^{fst}$ by time $GST{+}2\Delta$. With our pull protocol ensuring all live blocks can be acceptable within 2Δ, all validators in $\mathcal{V}_h^{slw}$ can accept all round r blocks from $\mathcal{V}_h^{fst}$ by $GST{+}4\Delta = t_r{+}\Delta$. Consequently, validators in $\mathcal{V}_h^{slw}$ can create their round $r{+}1$ blocks at time $t_r{+}\Delta$. In addition, because validators in $\mathcal{V}_h^{fst}$ can receive $\mathcal{V}_h^{slw}$'s round r blocks by time $t_r{+}\Delta$, they must be able to create their round $r{+}1$ blocks by time $t_r{+}\Delta$. The latency of round $r{+}1$ is Δ. For the next round $r{+}2$, there are two cases.

Case 1: If all honest validators enter round $r{+}1$ at time $t_r{+}\Delta$, according to Lemma 5, either the expected latency of round $r{+}2$ is within 2Δ or at least one malicious validator is blamed by honest validators.

Case 2: Otherwise, honest validators in $\mathcal{V}_h^{fst}$ enter round $r{+}1$ before they receive $\mathcal{V}_h^{slw}$'s round r blocks at $t_r + \Delta$. There are two scenarios, and we show that the expected latency of round $r{+}2$ is within 2Δ before at least one malicious validator is blamed.

First, every round r block created by malicious validators $\mathcal{A}$ is referenced by at least $f{+}1$ round $r{+}1$ blocks created by honest validators in $\mathcal{V}_h^{fst}$. In this scenario, thanks to the ImPoA-based pull mechanism, every honest validator in $\mathcal{V}_h^{slw}$ can accept these round $r{+}1$ blocks without synchronizing any missing blocks on the push path. As a result, $\mathcal{V}_h^{slw}$ can create their round $r{+}2$ blocks at time $t_r + 2\Delta$. Since, $\mathcal{V}_h^{slw}$ create their round $r{+}1$ blocks at time $t_r + \Delta$ (as mentioned above), the latency of round $r{+}2$ is within Δ.

Second, at least one round r block created by malicious validators $\mathcal{A}$ is referenced by fewer than $f{+}1$ round $r{+}1$ blocks created by honest validators in $\mathcal{V}_h^{fst}$. This means that at least one malicious validator V_m delays sharing its round r block B_m^r with some validators in $\mathcal{V}_{h1}^{fst} \subset \mathcal{V}_h^{fst}$, because otherwise, all honest validators in $\mathcal{V}_h^{fst}$ can receive enough round r blocks and enter round $r{+}1$ at the same time. However, all honest validators in $\mathcal{V}_{h1}^{fst}$ must be able to create round $r{+}2$ blocks by time $t_r{+}2\Delta$, since the delayed validators in $\mathcal{V}_h^{slw}$ create their round $r{+}1$ blocks at time $t_r + \Delta$ (as mentioned above). As a result, once receiving $\mathcal{V}_{h1}^{fst}$'s round $r{+}2$ blocks at time $t_r{+}3\Delta$, $\mathcal{V}_h^{slw}$ can invoke the pull protocol to fetch any missing round $r{+}1$ blocks within 2Δ, and create their round $r{+}3$ blocks by time $t_r{+}5\Delta$. Recall that $\mathcal{V}_h^{slw}$ create their round $r{+}1$ blocks at time $t_r + \Delta$. For $\mathcal{V}_h^{slw}$, there are two rounds $r{+}1$ and $r{+}2$ between time $t_r{+}\Delta$ and $t_r{+}5\Delta$. As a result, the average latency of these two rounds is $(t_r{+}5\Delta - t_r{+}\Delta)/2 = 2\Delta$.

By induction, we can see that for any future round $r'{>}r$, the latency of round r' is within 2Δ before any malicious validator V_m is blamed. The proof is done.

□

Lemma 7. *After GST, malicious validators $\mathcal{A}$ in Beluga can only delay the progress of the protocol in a bounded number of rounds with an expected latency higher than 2Δ every R_L rounds.*

Proof. Recall that a validator gets its reputation decreased by R_L by all honest validators if it is blamed, where $R_L \geq 1$ is a predetermined parameter. Without loss of generality, we assume right after GST, each malicious validator has its reputation R_m, and the lowest reputation of an honest validator is R_h.

According to Lemma 6, malicious validators $\mathcal{A}$ can delay the protocol for a round r with latency at most 4Δ, without getting blamed. We denote this period as $\mathcal{D}_1$, and use $|\mathcal{D}_1|$ to represent the number of rounds being delayed with the expected latency of more than 2Δ. $\mathcal{D}_1$ will increase the latency of the protocol by at most 3Δ before $\mathcal{A}$ get blamed.

After period $\mathcal{D}_1$, all honest validators can enter the same round at the same time. According to Lemma 5, $\mathcal{A}$ might still be able to delay some future rounds with the expected latency of more than 2Δ, but each of such round will lead to at least one malicious validator being blamed and getting lose of R_L points. Based on the reputation difference, we can derive that f malicious validators can delay the protocol for $(R_m - R_h) * f/R_L$ rounds before their reputations are lower than that of any honest validator. We denote this period as $\mathcal{D}_2$, and similarly, we have $|\mathcal{D}_2| = (R_m - R_h) * f/R_L$. According to Lemma 5, each of these rounds will increase the round latency by at most 2Δ. As a result, $\mathcal{D}_2$ will increase the latency of the protocol by at most $(R_m - R_h) * f/R_L * 2\Delta$.

After period $\mathcal{D}_2$, $\mathcal{A}$ have the reputation equal to the lowest reputation of an honest validator. Since according to Lemma 3, the honest validators will not get their reputations decreased after GST, $\mathcal{A}$ need to perform correctly to get their reputation increased by honest validators. In particular, to delay f rounds with the expected latency of more than 2Δ for each after $\mathcal{D}_2$, these f malicious validators $\mathcal{A}$ perform carefully without getting blamed for at least R_L rounds, during which the expected latency of each round will be 2Δ. We denote this period as $\mathcal{D}_3$. During $\mathcal{D}_3$, $\mathcal{A}$ delay f rounds with the expected latency of higher than 2Δ every R_L rounds. Thus, we have $|\mathcal{D}_3| = f$. According to Lemma 5, each round being delayed will increase the round latency by at most 2Δ. As a result, $\mathcal{D}_3$ will increase the latency of the protocol by at most $f * 2\Delta$ every R_L rounds.

By considering the above three periods, we can conclude that $\mathcal{A}$ can delay the protocol with the expected round latency higher than 2Δ for at most $|\mathcal{D}_1|+|\mathcal{D}_2|+|\mathcal{D}_3| = 1+(R_m-R_h)*f/R_L+f$ rounds, and the extra latency introduced by $\mathcal{A}$ is at most $(3\Delta+(R_m-R_h)*f/R_L*2\Delta+f*2\Delta)$ every R_L rounds. As f, R_m, R_h, and R_L are all constants, the proof is done. □

Finally, we have a proof for Theorem 5.

Proof for Theorem 5. According to Lemma 7, for every R_L rounds, malicious validators $\mathcal{A}$ can only increase the round latency by more than 2Δ for a bounded number of rounds. For these rounds, the total extra latency introduced is $(3\Delta+(R_m-R_h)*f/R_L*2\Delta+f*2\Delta)$ every R_L rounds. For the other rounds, the expected round latency is 2Δ. In other words, to create blocks for R_L rounds, the latency of the protocol is at most $R_L * 2\Delta+(3\Delta+(R_m - R_h) * f/R_L*2\Delta+f*2\Delta)$. As a result, the average round latency of Beluga is at most $(R_L*2\Delta+(3\Delta+(R_m-R_h)*f/R_L*2\Delta+f*2\Delta))/R_L = 2\Delta(1+\frac{3+2f}{2R_L}+\frac{(R_m-R_n)f}{2\Delta R_L^2})$. By setting a sufficiently large R_L, the average round latency can be arbitrarily close to 2Δ. The proof is done. □

Appendix B. The Pseudocode of the AC-based Push Protocol

Figure 8 provides a pseudocode for Beluga's push protocol. The key components consist of a reputation mechanism (lines 23-32) and an admission control (lines 1-22), both of which are detailed in Section 4.2.

Appendix C. Beluga Implementation

We implement Beluga in Rust within Mysticeti [11] by forking the Mysticeti codebase [16]. It leverages `tokio` [44] for asynchronous networking, utilizing raw TCP sockets for communication implementing a reliable point-to-point channels, necessary to correctly implement the distributed system abstractions without relying on any RPC frameworks. For cryptographic operations, it rely on `ed25519-consensus` [45] for asymmetric cryptography and `blake2` [46] for cryptographic hashing. To ensure data persistence and crash recovery, it employs a Write-Ahead Log (WAL) optimizing I/O operations through vectored writes [47], efficient memory-mapped files, and minimizes copies and serialization through `minibytes` [48].

By default, this Mysticeti implementation uses a traditional optimistic push followed by a random pull protocol (described in Section 2) we modify its block synchronizer module to use Beluga instead. Implementing our mechanism requires to add less than 200 LOC, and does not require any extra cryptographic tool.

In addition to regular unit tests, we inherited and utilized two supplementary testing utilities from the Mysticeti codebase. First, a simulation layer replicates the functionality of the `tokio` runtime and TCP networking. This simulated network accurately simulates real-world WAN latencies, while the `tokio` runtime simulator employs a discrete event simulation approach to mimic the passage of time. Second, a command-line utility (called *orchestrator*) which deploys real-world clusters of Beluga on machines distributed across the globe.

```
Variables:
  $R_L-$ The score decrease each time
  $TR_i[]-$ An array of reputations (indexed by validators)
  struct block $B$
    ...                                                    ▷ original fields
    $B.weaklinks$ - used to link blocks that $v_i$ has received and accepted but not referenced as parents
    $B.watermark[]$ - an array of the highest round numbers of all validators' blocks received by $v_i$
    $B.ancestors[]$ - an array of the highest round numbers of all validators' blocks reachable from $B$
  ▶ Call $block\_propose_i(B, r)$ to push a round $r$ block $B$
procedure create_new_block($r$, $\mathcal{B}^{r-1}$)    ▷ $\mathcal{B}^{r-1}$ is a list of the latest received blocks from all validators with round $\leq r-1$
    Initializes a block $B$ with $B.r = r$, $B.author = i$, and other original fields
    $parents \leftarrow$ AC_parent_selection($r$, $\mathcal{B}^{r-1}$)
    $B.parents \leftarrow$ digests of $parents$
    $B.weaklinks \leftarrow \{B'.d | B' \in \mathcal{B}^{r-1} \backslash parents$ is acceptable$\}$
    $watermark \leftarrow []$
    for $\forall B' \in \mathcal{B}^{r-1}$ do
        $watermark[B'.author] \leftarrow B'.r$
    $B.watermark \leftarrow watermark$
    $B.ancestors \leftarrow$ compute_ancestors($parents$)
    signs and broadcasts $B$ using best-effort broadcast
    update_score_with_watermarks($r$, $\mathcal{B}^{r-1}$)
    outputs $block\_accept_i$ and $block\_store_i$ for $B$ and every acceptable block in $\mathcal{B}^{r-1}$, if it hasn't done so already
procedure AC_parent_selection($r$, $\mathcal{B}$)
    $\mathcal{B} \leftarrow \{B' \in \mathcal{B} | B'.r = r-1 \wedge B'$is acceptable$\}$
    $parents \leftarrow$ top $2f+1$ blocks in $\mathcal{B}$ by $TR_i[B'.author]$
    return $parents$
procedure compute_ancestors($parents$)
    $ancestors \leftarrow []$
    for $k \in 1, \cdots, n$ do
        $ancestors[k] \leftarrow \max(\{B'.r | B' \in parents \wedge B'.author = v_k\} \cup \{B'.ancestors[k] | B' \in parents\})$
    return $ancestors$
procedure update_score_with_watermarks($r$, $\mathcal{B}$)
    for $j \in 1, \cdots, n$ do
        $count \leftarrow 0$
        for $\forall B' \in \mathcal{B}$ do
            if $B'.watermark[j] == r-2$ then
                $count \leftarrow count + 1$
        if $count \geq 2f+1$ then
            $TR_i[j] \leftarrow TR_i[j] + 1$
upon pulling or receiving $f+1$ pull requests (i.e., blames) for a missing block created by $v_j$ do
    $TR_i[j] \leftarrow TR_i[j] - R_L$
```

Figure 8. Beluga's AC-based optimistic push protocol for validator v_i.

We are open-sourcing our Beluga implementation, along with its simulator and orchestration tools, to ensure reproducibility of our results[2].

Appendix D. Experimental Setup

This section describes the experimental setup used for evaluating Beluga in Section 6.

We deploy all systems on AWS, using `m5d.8xlarge` instances across 13 different AWS regions: Northern Virginia (us-east-1), Oregon (us-west-2), Canada (ca-central-1), Frankfurt (eu-central-1), Ireland (eu-west-1), London (eu-west-2), Paris (eu-west-3), Stockholm (eu-north-1), Mumbai (ap-south-1), Singapore (ap-southeast-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), and Seoul (ap-northeast-2). Validators are distributed across those regions as equally as possible. Each machine provides 10 Gbps of bandwidth, 32 virtual CPUs (16 physical cores) on a 3.1 GHz Intel Xeon Skylake 8175M, 128 GB memory, and runs Linux Ubuntu server 24.04. We select these machines because they provide decent performance, are in the price range of "commodity servers", and match the minimal specifications of modern quorum-based blockchains [49].

The *latency* refers to the time elapsed from the moment a client submits a transaction to when it is committed by the validators, and the *throughput* refers to the number of transactions committed per second. We instantiate several geo-distributed benchmark clients within each validator submitting transactions in an open loop model, at a fixed rate. We experimentally increase the load of transactions sent to the systems, and record the throughput and latency of commits. As a result, all plots illustrate the steady-state latency of all systems under various loads. Transactions in the benchmarks are arbitrary and contain 512 bytes. We configure both systems with 2 leaders per round, and a leader timeout of 1 second.

[2] https://github.com/asonnino/beluga/tree/paper (commit `9a3d2a3`)