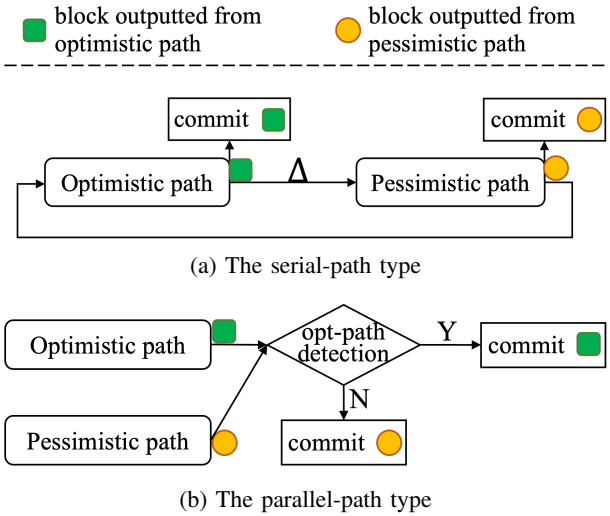
Icarus: Achieving Performant Asynchronous BFT with Only Optimistic Paths

Xiaohai Dai*, Yiming Yu*, Sisi Duan†✉, Rui Hao‡✉, Jiang Xiao*, and Hai Jin*
*National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology
†Tsinghua University, State Key Laboratory of Cryptography and Digital Economy Security
‡School of Computer Science and Artificial Intelligence, Wuhan University of Technology
{xhdai, yming_yu}@hust.edu.cn, duansisi@tsinghua.edu.cn, ruihao@whut.edu.cn, {jiangxiao, hjin}@hust.edu.cn
✉Corresponding authors

***Abstract*—The emergence of blockchain technology has revitalized research interest in *Byzantine Fault Tolerant* (BFT) consensus, particularly asynchronous BFT due to its resilience against network attacks. To improve the performance of traditional asynchronous BFT, recent studies propose the dual-path paradigm: an optimistic path for efficiency under favorable situations and a pessimistic path—typically implemented through a *Multi-valued Validated Byzantine Agreement* (MVBA) protocol—to guarantee liveness in unfavorable situations. However, owing to the inherent complexity and inefficiency of the MVBA protocol, existing dual-path protocols exhibit high implementation complexity and poor performance in unfavorable situations. Moreover, the two constituent types within the dual-path paradigm—serial-path and parallel-path—each face additional limitations. Specifically, the serial-path type encounters difficulties in switching between the optimistic and pessimistic paths, whereas the parallel-path type discards blocks from one of the paths, resulting in bandwidth waste and reduced throughput.**

To address these limitations, we propose Icarus, a single-path asynchronous BFT protocol that exclusively leverages optimistic paths without pessimistic paths. The optimistic path ensures Icarus's efficiency under favorable situations. To guarantee liveness in unfavorable conditions, Icarus employs a rotating-chain mechanism: each node broadcasts a chain of blocks in parallel, and these chains take turns serving as the optimistic path in a round-robin fashion. Since non-faulty nodes' chains continuously grow, once a chain accumulating enough blocks becomes the optimistic path, its blocks can be committed, ensuring liveness even in unfavorable conditions. To maintain consistency during path transitions, Icarus introduces the *Two-consecutive-validated-value Byzantine Agreement* (tcv^2-BA) protocol, which aligns heights of committed blocks on the previous path. We have verified Icarus's correctness through theoretical analysis and validated its high performance through various experiments.



(a) The serial-path type

(b) The parallel-path type

Fig. 1: Schematic diagram of the dual-path paradigm

I. Introduction

A. Dual-path asynchronous BFT and its issues

1) Asynchronous BFT: The emergence of blockchain technology has sparked widespread interest in *Byzantine Fault Tolerant* (BFT) consensus protocols [1], [2]. These protocols play a crucial role in maintaining data consistency across distributed nodes, even in the presence of malicious (Byzantine) nodes [3], [4]. Traditional BFT protocols are broadly classified into three categories based on network assumptions [5], [6]: synchronous, partially synchronous, and asynchronous. Synchronous protocols [7], [8] rely on strong network assumptions that are difficult to satisfy in real-world deployments, compromising their practical applicability. Partially synchronous protocols [9], [10] offer improved performance but remain vulnerable to liveness violations under network attacks [11]. Consequently, recent research has focused on the asynchronous protocols [12], [13], which exhibit the highest robustness by making no timing assumptions. However, these traditional asynchronous protocols typically achieve significantly lower performance than their partially synchronous counterparts.

2) Dual-path asynchronous BFT: To improve the performance of asynchronous BFT protocols, a line of studies has proposed the dual-path paradigm that combines an optimistic path and a pessimistic path [14], [15], [16], [17]. The op-

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240060
www.ndss-symposium.org

timistic path is built upon partially synchronous protocols such as two-phase HotStuff [10], while the pessimistic path is implemented using a purely asynchronous protocol—typically *Multi-valued Validated Byzantine Agreement* (MVBA) [18], [19]. Under the favorable conditions, the dual-path protocol commits blocks through the optimistic path, achieving high performance. Conversely, when it is in an unfavorable situation, the dual-path protocol falls back to the pessimistic path to ensure liveness.

Based on the coordination strategy between the two paths, existing dual-path protocols can be broadly categorized into two types: serial-path and parallel-path. As illustrated in Fig. 1a, serial-path protocols execute the two paths sequentially, relying on timeout mechanisms to determine when to transition from the optimistic path to the pessimistic one [14], [15]. The protocol resumes using the optimistic path only after a predefined number of blocks have been successfully committed via the pessimistic path.

In contrast, parallel-path protocols execute both paths concurrently [16], [17], as shown in Fig. 1b. Within this type, a detection mechanism is employed to assess the effectiveness of the optimistic path. Under favorable conditions, the detection mechanism signals that the optimistic path is performing smoothly, leading the protocol to commit blocks from the optimistic path while discarding those from the pessimistic path. Conversely, in unfavorable conditions, the mechanism detects the failure of the optimistic path, prompting the protocol to commit blocks from the pessimistic path instead.

3) Issues of dual-path BFT: However, existing dual-path protocols exhibit several fundamental limitations. First, in both serial-path and parallel-path types, the pessimistic path depends on MVBA protocols, whose intrinsic complexity significantly amplifies the overall complexity of the dual-path architecture, which in turn hinders practical implementation. Moreover, the inherent inefficiency of MVBA degrades the performance of dual-path protocols. Specifically, each instance of MVBA can only commit a single block, imposing a bottleneck on throughput. Furthermore, cutting-edge protocols like sMVBA [19] require at least six communication rounds to commit a block, while FIN suffers from cubic communication complexity [20]. Second, serial-path protocols rely heavily on precise tuning of timeout parameters to facilitate timely transitions between optimistic and pessimistic paths; improper configuration can lead to pronounced performance degradation. Third, even in favorable situations, parallel-path protocols continue executing the pessimistic path while only committing blocks from the optimistic path. This leads to significant resource waste and low throughput, as the pessimistic path is redundantly executed, yet its outputs are discarded.

B. Our solution: asynchronous BFT with only optimistic paths

To address the aforementioned issues, we propose Icarus, an efficient single-path asynchronous BFT protocol that operates exclusively on optimistic paths. By removing pessimistic paths, Icarus simplifies the protocol design and enhances practical implementability. In the event of an optimistic path failure, Icarus can promptly switch to an alternative optimistic path without reverting to a pessimistic one, thus improving performance. Furthermore, by employing a single type of paths, Icarus circumvents the difficulties associated with switching between optimistic and pessimistic paths, which are inherent in serial-path protocols. Compared to parallel-path protocols, Icarus guarantees that all broadcasted blocks are eventually committed, avoiding resource wastage and further improving overall throughput.

In fact, designing an asynchronous protocol using only optimistic paths introduces significant challenges in liveness, safety, and efficiency. Icarus overcomes these challenges through three key components, as outlined below.

1) Parallel chains with path rotation to guarantee liveness: **Liveness challenge—**An optimistic path may be disrupted by the adversary through deliberately delaying blocks proposed by the leader, thus compromising the liveness property.

Solution. To address this, we introduce a novel approach that combines parallel chains with rotating optimistic paths. Specifically, each node continuously broadcasts its chain of blocks, with each block embedding the *Quorum Certificate* (QC) of its immediate predecessor. Each chain essentially constitutes the two-phase HotStuff structure, excluding the view-change mechanism [10]. We say a block references another block if the former contains the QC for the latter. These individual chains take turns serving as the optimistic path, based on a deterministic rotation schedule. The block committing process on the optimistic path follows the two-phase HotStuff protocol [21], [22], in which the receipt of a block at height h enables the committing of the block at height $h-2$. As all non-faulty nodes constantly extend and broadcast their chains, block committing is triggered whenever a chain accumulates more than two uncommitted blocks and becomes the optimistic path, thereby ensuring liveness.

2) Leveraging a binary agreement variant to ensure safety: **Safety challenge—**Due to network asynchrony, nodes may initiate path switching at different block heights, potentially leading to inconsistencies in the number of blocks committed from the previous optimistic path. Such divergence can compromise the safety property of the protocol.

Solution. To resolve this, Icarus introduces the *two-consecutive-validated-value Byzantine Agreement* (tcv^2-BA)—a customized variant of the *Asynchronous Binary Agreement* (ABA) [23] or an extension of *two-consecutive-value Byzantine Agreement* (tcv-BA) [14]. Icarus first conducts a height-exchange round to align the locked block heights to one of two adjacent values. It then leverages tcv^2-BA to reach agreement on the final committing height. This process ensures that all non-faulty nodes commit consistent blocks of the previous optimistic path, thereby preserving the safety.

3) Inter-chain references for high throughput and adaptive path switching: **Efficiency challenge**—In a parallel-chain structure, committing only blocks from the optimistic path may result in the wasted dissemination of blocks from non-optimistic-path chains.

Solution. To mitigate this inefficiency, each block in Icarus not only references its predecessor within the same chain (intra-chain references), but also references blocks from other chains (inter-chain references). Based on these inter-chain references, committing a block on the optimistic path also triggers the committing of referenced blocks from other chains, thereby improving throughput.

When the current optimistic path functions smoothly, blocks from all chains can be continuously committed, keeping the number of uncommitted blocks in each chain low. Based on this observation, Icarus incorporates an adaptive path-switching mechanism: once any non-optimistic-path chain accumulates a threshold of uncommitted blocks, this indicates the failure of the current optimistic path, prompting the system to initiate a path rotation.

4) Experimental evaluation: We implement a prototype of Icarus and evaluate its performance against four baseline protocols: Ditto, ParBFT, sMVBA [19], and Tusk [24]. The evaluation is conducted under two situations. In favorable situations, optimistic leaders operate without artificial delays, allowing blocks to be committed via the optimistic path. In unfavorable situations, artificial delays are injected into the optimistic leaders to trigger path switching. The experimental results demonstrate that Icarus achieves minimal latency under favorable situations by leveraging optimistic paths. Even in unfavorable situations, Icarus maintains superior latency compared to baselines, as it can immediately commit blocks on the newly activated optimistic path after switching, effectively reducing transition overhead. Furthermore, Icarus consistently outperforms all baselines in throughput across both situations, owing to its parallel-chain structure and inter-chain references.

II. Background and Preliminaries

A. Model & assumptions

We consider a system consisting of n nodes, uniquely identified by indices p_i where $0 \leq i < n$. Among these nodes, f nodes may deviate from protocol execution (satisfying the condition $3f+1 \leq n$) and are referred to as Byzantine nodes. The remaining nodes are termed non-faulty nodes. Like most BFT protocols [10], [14], [16], [17], the set of nodes must be fixed prior to the system initiation, which does not allow for dynamic joining or replacement. We assume the existence of an adversary that can coordinate all Byzantine nodes to perform malicious actions, thereby exerting its maximum capability to compromise either the safety or liveness of the consensus protocol. Nodes are interconnected via a point-to-point network. To ensure the protocol's tolerance against network attacks, we assume the network is asynchronous. Specifically, the adversary can arbitrarily delay message transmissions between non-faulty nodes, provided that all messages eventually reach their intended destinations.

We assume the system has established a *Public Key Infrastructure* (PKI) and a threshold signature scheme. The PKI mechanism guarantees the non-repudiation and integrity of messages sent by nodes, while the threshold signature scheme enables the construction of succinct signatures from a group of nodes. Additionally, we assume the existence of a cryptographic hash function. We further assume the adversary is computationally bounded, meaning it cannot compromise the security of the PKI, the threshold signature scheme, or the hash function.

B. Asynchronous BFT & SMR

1) Definition of BFT and SMR: The *Byzantine Fault Tolerant* (BFT) consensus is used to maintain data consistency among a group of distributed nodes even when some nodes act maliciously [25]. Typically, the BFT consensus can be employed to implement *State Machine Replication (SMR)*. In this paper, we primarily focus on designing BFT consensus protocols in the context of SMR. Each node p_i maintains a local block vector denoted as $\mathcal{V}_i$, where each element represents a block and is indexed as $\mathcal{V}_i[k]$ ($k \geq 0$). Blocks that have undergone BFT consensus are written into $\mathcal{V}_i$. The vector $\mathcal{V}_i$ follows an append-only property, meaning that $\mathcal{V}_i[k]$ can only be written when for all $m < k$, $\mathcal{V}_i[m] \neq \perp$ and $\mathcal{V}_i[k] = \perp$.

Each node maintains a transaction buffer to receive transactions from clients. When generating new blocks, nodes select the earliest transactions from the buffer (with the number of transactions limited by block size). Once a block is committed through BFT consensus, its contained transactions are removed from the buffer. If a block is committed, all transactions it contains are considered committed. As a widely adopted practice [9], [26], when clients input new transactions, they first send the transaction to a specific node. If the transaction cannot be committed within a certain period, the client will broadcast it to all nodes.

A correctly implemented SMR based on BFT must satisfy two key properties:

- **Safety:** For any two non-faulty nodes p_i and p_j, if $\mathcal{V}_i[k] \neq \perp$ and $\mathcal{V}_j[k] \neq \perp$, then $\mathcal{V}_i[k] = \mathcal{V}_j[k]$.
- **Liveness:** If a transaction is added to the buffer of every non-faulty node, it will eventually be committed.

2) Asynchronous BFT: BFT consensus protocols can be classified into three categories based on network assumptions: synchronous, partially synchronous, and asynchronous protocols. Among these, asynchronous protocols have received significant attention in recent years due to their minimal network assumptions, enabling tolerance against stronger network attacks. However, compared to partially synchronous protocols, traditional asynchronous protocols typically incur higher communication overhead and larger latency.

C. Asynchronous BFT with dual paths

To enhance the performance of asynchronous BFT, several approaches have introduced optimistic paths while employing purely asynchronous protocols as the pessimistic path, forming what is known as dual-path protocols [14], [15], [16], [17]. Typically, the optimistic path is implemented based on partially synchronous protocols. When the system operates under favorable conditions, dual-path protocols can commit blocks through the optimistic path, significantly improving efficiency.

Conversely, when facing unfavorable network conditions, these protocols can rely on the pessimistic path to ensure liveness.

Based on the operational patterns of the two paths, dual-path protocols can be categorized into serial-path and parallel-path types. The serial-path type executes the two paths sequentially [14], [15]: it first attempts the optimistic path, and only invokes the pessimistic path when the optimistic one fails. After committing a certain number of blocks via the pessimistic path, the system reverts to attempting the optimistic path. In contrast, the parallel-path type runs both paths concurrently [16], [17], committing blocks from the optimistic path when conditions are favorable or from the pessimistic path when conditions are unfavorable.

Motivation. However, dual-path protocols exhibit several limitations. First, the pessimistic paths typically employ MVBA protocols [18], [19], whose structural complexity amplifies the overall intricacy of the dual-path protocols, thereby complicating implementation. In addition, MVBA's limited efficiency further impairs the performance under unfavorable situations. The throughput of existing MVBA protocols is fundamentally constrained, as each instance commits only one block. They also impose a difficult trade-off: achieving low communication complexity leads to large latency (e.g., six rounds in sMVBA [19]), while reducing latency comes at the cost of cubic communication complexity (as in FIN [20]). Second, while optimistic paths excel in favorable conditions, pessimistic paths are better suited for unfavorable conditions. This necessitates timely switching between optimistic and pessimistic paths to avoid excessive latency in the dual-path protocols. Finally, parallel-path protocols waste resources by running both paths simultaneously while only committing blocks from one path, leading to inefficient resource utilization and low throughput.

D. Two-consecutive-value Byzantine agreement

The *two-consecutive-value Byzantine Agreement* (tcv-BA) protocol, proposed by Lu et al. [14], can be viewed as a generalization of the *Asynchronous Binary Agreement* (ABA) protocol [27], [28]. In tcv-BA, both inputs and outputs are integer values. All non-faulty nodes' input values belong to a consecutive two-integer set, specifically $S = \{v, v+1\}$, where each non-faulty node p_i's input value $in_i \in S$. All non-faulty nodes ultimately output a consistent integer value through tcv-BA. Formally, a correct tcv-BA protocol must satisfy the following properties:

- **Agreement:** For any two non-faulty nodes p_i and p_j, if they output out_i and out_j respectively from tcv-BA, then $out_i = out_j$.
- **Validity:** If any non-faulty node outputs out, then at least one non-faulty node takes out as the input value.
- **Termination:** If all non-faulty nodes activate the tcv-BA protocol, then eventually all non-faulty nodes will output a value.

The tcv-BA protocol can be constructed based on any existing ABA protocol. Due to space limitations, we omit the specific construction details of tcv-BA, which can be found in Section 5.2 of the BDT paper [14].

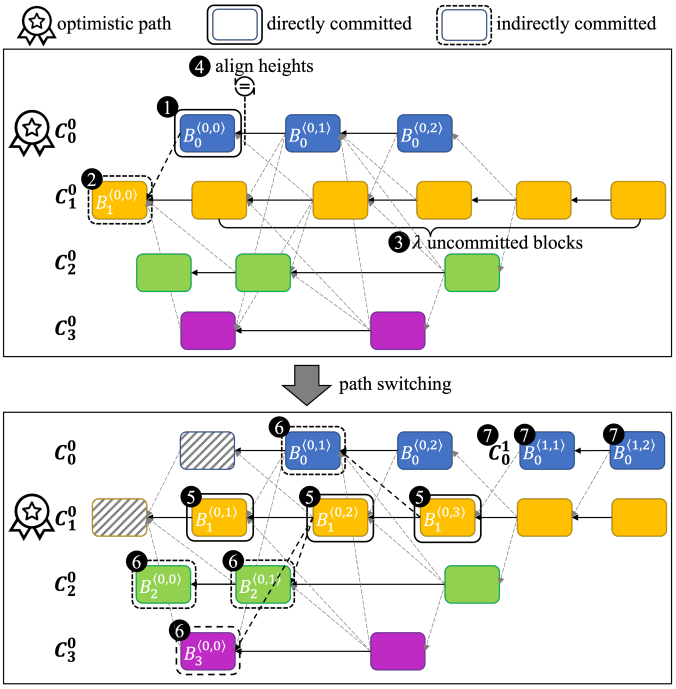


Fig. 2: The schematic diagram of Icarus, with color-filled blocks to differentiate those generated by distinct nodes for enhanced readability

III. Icarus Design

Driven by the motivations outlined in Section II-C, we introduce Icarus, a novel asynchronous BFT protocol that operates exclusively on optimistic paths.

A. Overview

At a high level, each node broadcasts a chain following the two-phase HotStuff protocol, with the view-change mechanism removed. For brevity, we refer to a chain broadcast by p_i as p_i's chain. These chains take turns serving as the optimistic path that drives the consensus process. When the current optimistic path fails, a path switch is triggered, and the next chain is selected as the new optimistic path. To ensure safety, an alignment protocol is invoked during the path switch to synchronize the block heights committed on the previous optimistic path. Furthermore, to improve performance, each block in Icarus can also incorporate QCs for blocks from other chains. As a result, committing a block on the optimistic path can indirectly trigger the committing of blocks on the other chains through these QC links.

Additionally, after a path switch, the broadcaster associated with the previous optimistic path may resume broadcasting blocks (possibly due to network recovery or node fault recovery), thereby forming a new chain. To distinguish between multiple chains generated by the same node, we assign an epoch number to each chain. Thus, a block in Icarus is denoted

as $B_i^{\langle e,h\rangle}$, where i is the broadcaster's identifier, e is the epoch number (starting from 0), and h is the block's height within this chain (also starting from 0). Correspondingly, each chain is represented as C_i^e, and a block at height h within this chain is denoted as $C_i^e[h]$.

It should be noted that, by including QCs for blocks from other nodes, Icarus exhibits a topological similarity to DAG-based protocols such as DAGRider [29] and Tusk [24]. However, the two approaches differ significantly. Specifically, existing DAG-based protocols mandate that block generation across different nodes maintain a comparable frequency—each new block must wait and reference at least $n-f$ previously generated blocks. This requirement inherently constrains the block generation rate and system throughput. In contrast, Icarus does not enforce the constraint that a new block must reference $n-f$ prior blocks, thereby allowing different nodes to generate blocks at their own pace. This flexibility leads to an improvement in throughput, a claim that will be empirically validated in the experimental section (Section V).

Illustration through an example. Fig. 2 illustrates the architecture of Icarus, which consists of four nodes. For example, Blocks $B_0^{\langle 0,0\rangle}$, $B_0^{\langle 0,1\rangle}$, and $B_0^{\langle 0,2\rangle}$ form a chain C_0^0. $B_0^{\langle 0,1\rangle}$ not only references $B_0^{\langle 0,0\rangle}$, but also references $B_1^{\langle 0,1\rangle}$. Fig. 2 begins with chain C_0^0 serving as the optimistic path. Upon receiving block $B_0^{\langle 0,2\rangle}$, block $B_0^{\langle 0,0\rangle}$ is committed (as shown in ❶ of Fig. 2) according to the two-phase commit rule [21], [22]. Since $B_0^{\langle 0,0\rangle}$ references block $B_1^{\langle 0,0\rangle}$, the latter is also committed (as shown in ❷ of Fig. 2). To distinguish between these cases, we refer to the commit of a block based on the two-phase commit rule as a direct commit (e.g., block $B_0^{\langle 0,0\rangle}$), while the commit of a block triggered by the commit of another block is called an indirect commit (e.g., block $B_1^{\langle 0,0\rangle}$).

When the number of uncommitted blocks in other chains reaches λ (as shown by C_1^0 in ❸), the optimistic path will be switched. To ensure consistency in the number of committed blocks on the previous optimistic path, Icarus introduces the tcv^2-BA protocol to align the heights of committed blocks (❹). Subsequently, the optimistic path switches to the next chain, as illustrated by C_1^0 in Fig. 2. On the new optimistic path, if the number of uncommitted blocks exceeds two, all blocks except the last two are directly committed (as shown in ❺ for $B_1^{\langle 0,1\rangle}$, $B_1^{\langle 0,2\rangle}$, and $B_1^{\langle 0,3\rangle}$). These blocks also trigger indirect commits of $B_2^{\langle 0,0\rangle}$, $B_2^{\langle 0,1\rangle}$, $B_3^{\langle 0,0\rangle}$, and $B_0^{\langle 0,1\rangle}$ in ❻. Furthermore, when a node p_i rebroadcasts a block, it generates a new chain with its epoch number incremented by one (as shown in ❼ for C_0^1 in Fig. 2).

As described above, Icarus is entirely composed of optimistic paths and does not require any pessimistic path execution, thereby avoiding the various limitations associated with pessimistic paths implemented through MVBA. Ideally, it is desirable for Icarus to continuously commit blocks using the current chain as the optimistic path, without triggering any path switches. Based on this perspective, we formalize the notions of favorable and unfavorable situations. A favorable situation occurs when the system can commit blocks along the current optimistic path without requiring a path switch. Conversely, any situation that does not meet this condition is classified as unfavorable.

Algorithm 1: Data structures & utilities for p_i

1 Let D_j denote the last certified block in C_j^ν where C_j^ν is the latest chain generated by p_j.
2 **struct** Block $\{i, e, h, sqc, wqc, d\}$
3 **define** *GenBlk(e, h, sqc)*:
4 $d \leftarrow$ a batch of transactions from $\texttt{buf}_i$;
5 $B.e \leftarrow e$; $B.h \leftarrow h$; $B.d \leftarrow d$; $B.sqc \leftarrow sqc$;
6 **update** B's references;
7 **foreach** $j \in [1..n]$ ***and*** $j \neq i$ **do**
8 **if** D_j *is not referenced by* B **then**
9 $B.wqc \leftarrow B.wqc \cup \{\text{QC for } D_j\}$;
10 **update** B's references;
11 **return** B

In the remainder of this section, we detail the individual components of the Icarus protocol, encompassing its data structures, the tcv^2-BA protocol, the block committing rule, and the path switching rule. To facilitate a rigorous and clear presentation, several algorithms are introduced. A complete execution example of these algorithms is illustrated to enhance reader comprehension; however, owing to space constraints, this illustration is deferred to Appendix A.

B. Data structures & chain generation

1) Definition of references: Each block contains a QC for its predecessor on the same chain, as well as multiple QCs for blocks on the other chain. We refer to the former as a strong reference and the latter as a weak reference. If a QC is generated for block B, the height of this QC is defined as the same as the height of B. A QC comprises the hash of the referenced block along with $n-f$ distinct votes from different nodes. These votes can be aggregated either through a threshold signature scheme or by combining $n-f$ PKI signatures. The reference relationship exhibits transitivity: if block A references block B, and block B references block C, then block A is considered to transitively reference block C. All blocks referenced by block A are called the ancestors of A. Specifically, every block is inherently considered its own ancestor.

2) Definition of block: As shown in Algorithm 1, each block $B_i^{\langle e,h\rangle}$ in Icarus comprises six fields. The first three fields (i, e, and h) serve as unique identifiers for the block. The remaining fields maintain two types of references: a strong reference (sqc) and weak references (wqc). The block also contains a transaction batch d for execution. A block is considered certified when the QC for it has been generated. During the block generation for B, the last certified block D_j from every other chain will be checked (Line 7 of Algorithm 1). If D_j has not been referenced by B, the QC for D_j will be

Algorithm 2: Generation of the chain C_i^e

Let $conc$ denote whether C_i^e is concluded.
Let $QC_i^{\langle e,h\rangle}$ denote QC for block $B_i^{\langle e,h\rangle}$.
define $NewChain(i,\ e,\ conc)$**:**
  $h \leftarrow 0$;
  while *not* $conc$**:**
    if $h > 0$ **then**
      wait until $QC_i^{\langle e,h-1\rangle}$ is generated;
      $sqc \leftarrow QC_i^{\langle e,h-1\rangle}$;
    else
      $sqc \leftarrow \perp$;
    $B_i^{\langle e,h\rangle} \leftarrow GenBlk(e,h,sqc)$;
    broadcast $B_i^{\langle e,h\rangle}$;
    h++;
on *receiving* $B_j^{\langle e,h\rangle}$ *from* p_j**:**
  append $B_j^{\langle e,h\rangle}$ to the chain C_j^e;
  send vote on $B_j^{\langle e,h\rangle}$ to p_j;
on *receiving* $n-f$ *votes on* $B_i^{\langle e,h\rangle}$**:**
  generate $QC_i^{\langle e,h\rangle}$ based on these votes;

incorporated into $B.wqc$ (Lines 8-10 of Algorithm 1). The reference set of B undergoes updates whenever a new QC is added to the block, as evidenced by the operations in Lines 6 and 10. Through the sqc and wqc fields, each block must newly reference at least f blocks from other distinct chains compared to its immediate predecessor.

3) Generation of a chain: In Icarus, except for the genesis block of the chain, a block creator can only construct and broadcast the next block after receiving the QC for the preceding block. As illustrated in Lines 5-10 of Algorithm 2, the sqc field of the genesis block is set to $\perp$, while for subsequent blocks, sqc contains the QC for the immediately preceding block. Upon receiving a new block, each node verifies its validity. If the node has not yet delivered any ancestor block of the new block, it initiates a block retrieval request to the creator of the new block. The node will only deliver the new block after all its ancestor blocks have been delivered. For brevity, Algorithm 2 omits the detailed description of block verification and retrieval procedures. Once the verification passes, the node casts its vote on the block (Line 16). The vote is a partial threshold signature (for threshold-based QC) or a PKI signature (for PKI-based QC). After collecting $n-f$ votes, the block creator aggregates these votes to generate the QC for the block (Lines 17-18).

C. Two-consecutive-validated-value Byzantine agreement

The tcv^2-BA protocol extends tcv-BA by incorporating the external validity property. Its inputs consist of two parameters, h and qc, where h retains the same meaning as in tcv-BA, while qc is used for external validation. The output is a consistent v value.

Algorithm 3: Π_{tcvv}^{id} for the node p_i with id identifying the tcv^2-BA instance

Let Π_{tcv}^{id} denote the tcv-BA instance.
define $\Pi_{tcvv}^{id}(h, qc)$**:**
  broadcast $\langle h, qc\rangle$ in the `1P` message;
  on *receiving* `1P` *message from* p_j**:**
    $\langle h_m, qc_m\rangle \leftarrow$ data contained in this message;
    $blk \leftarrow$ the block qc_m refers to;
    ▷ external validation
    if *any ancestor block of* blk *is not delivered* **then**
      retrieve blk's ancestors from p_j recursively;
    ▷ next, process h_m as v in Π_{tcv}^{id}
  ▷ the rest remains the same as Π_{tcv}^{id}
  on *returning* v_o *from* Π_{tcv}^{id}**:**
    return v_o;

The construction of tcv^2-BA, denoted as Π_{tcvv}, is presented in Algorithm 3 and is derived by modifying any existing tcv-BA construction Π_{tcv} [14]. Specifically, in the first broadcast round, it broadcasts a tuple $\langle h, qc\rangle$ (Line 3). Upon receiving this tuple, nodes perform external validation based on qc. During external validation, nodes check whether all ancestor blocks referenced by qc have been delivered. If any ancestor remains undelivered, they recursively retrieve it from the corresponding node (Lines 4-8). The processing of the h component in this tuple follows the same procedure as the v value in Π_{tcv}. Furthermore, all subsequent rounds of Π_{tcvv} remain identical to those in Π_{tcv}. Once the v_o value is returned from the Π_{tcv} instance, it is directly adopted as the output of Π_{tcvv} (Lines 9-10).

With the incorporation of random common coins in its tcv^2-BA (more specifically, tcv-BA) protocol, Icarus can circumvent the FLP impossibility theorem [30], thereby achieving asynchronous consensus.

D. Block committing through a path

The commit process of blocks essentially involves writing the blocks into the underlying vector $\mathcal{V}_i$, which requires performing a global sorting of these blocks.

1) Block committing: As described in Section III-A, block committing in Icarus follows two forms: direct committing and indirect committing. The direct committing process operates through either the two-phase commit rule or path finalization. For two-phase rule-based direct committing, as illustrated in Lines 9-11 of Algorithm 4, C_{oid}^{ν} denotes the chain corresponding to the current optimistic path, where oid represents the rotating node identifier for the optimistic path and ν indicates the epoch number of the latest chain generated by p_{oid}. When an uncommitted block in C_{oid}^{ν} possesses two successor blocks, it triggers direct committing. Path-finalization-based direct committing will be detailed in Section III-E1. The direct committing operation subsequently induces the indirect committing of its ancestor blocks.

Algorithm 4: Icarus protocol for p_i

1 **Let** Γ record the latest chain epoch of each node.

2 **define** *main()*:
3 $oid \leftarrow 0$; $\Gamma[:] \leftarrow 0$;
4 $conc_0 \leftarrow$ `false`; $NewChain(i, 0, conc_0)$;
5 **while** `true`:
6 $switch \leftarrow$ `false`; $\nu \leftarrow \Gamma[oid]$;
7 $C^{\nu}_{oid} \leftarrow$ the chain with epoch ν generated by p_{oid};
8 **while** *not switch*:
9 $h_l \leftarrow$ height of last uncommitted block in C^{ν}_{oid};
10 **if** *there are two or more blocks after* h_l **then**
11 $GlobalSort(C^{\nu}_{oid}[h_l])$;
12 **if** $oid = i$ **then**
▷ initiate a new chain
13 $conc_{\nu} \leftarrow$ `true`;
14 $conc_{\nu+1} \leftarrow$ `false`;
15 $NewChain(i, \nu+1, conc_{\nu+1})$;
16 $\Gamma[oid] \leftarrow \nu + 1$;
17 $oid \leftarrow oid + 1 \mod n$;
▷ trigger the path switch
18 **on** *#uncommitted blocks in* $C^{\Gamma[k]}_{k} \geq \lambda$:
19 $switch \leftarrow$ `true`;
20 **stop** voting for blocks on C^{ν}_{oid};
21 **invoke** $\Pi^{\langle oid,\nu\rangle}_{align}$;
▷ finalize the previous path
22 **on** *returning* h_{ba} *from* $\Pi^{\langle oid,\nu\rangle}_{align}$:
23 **foreach** $h \leq h_{ba}$ ***and*** $C^{\nu}_{oid}[h]$ *is uncommitted* **do**
24 $GlobalSort(C^{\nu}_{oid}[h])$;

Algorithm 5: Global sorting protocol (for p_i)

1 **define** $GlobalSort(B)$:
2 $S_{com} \leftarrow$ all committed blocks;
3 $S_{anc} \leftarrow$ ancestor blocks of B;
4 $S_{toc} \leftarrow S_{anc} \setminus S_{com}$;
5 $S^{*}_{toc} \leftarrow$ **sort** S_{toc} first by the block creator's identifier, then by epoch, and finally by height;
6 **append** S^{*}_{toc} to $\mathcal{V}_i$;

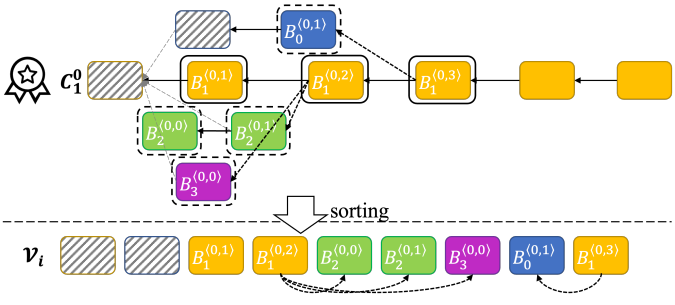


Fig. 3: An example to demonstrate the global sorting rule. We omit some references in the figure for brevity.

Algorithm 6: $\Pi^{\langle i,e\rangle}_{align}$ where $\langle i, e\rangle$ identifies the path

1 $B_{cert} \leftarrow$ the last certified block in the chain C^{e}_{i};
2 **broadcast** QC for B_{cert} in the `QC-EX` message;
3 **on** *receiving* $n - f$ *QC-EX messages*:
4 $QC_l \leftarrow$ QC with the largest height in these messages;
5 $h_{qc} \leftarrow QC_l$'s height;
6 $h_{ba} \leftarrow \Pi^{\langle i,e\rangle}_{tcvv}(h_{qc}, QC_l)$;
7 **return** h_{ba};

2) Global block sorting: It should be noted that we do not explicitly perform the indirect commit for blocks in Algorithm 4. In fact, whether it is a direct commit or an indirect commit, the corresponding block is globally sorted and written into $\mathcal{V}_i$. Thus, the global sorting process implicitly accomplishes the indirect commit, with the specific rules defined in the *GlobalSort* function of Algorithm 5. Specifically, if a block B is directly committed, we first remove all already-committed ancestor blocks from B's ancestor set, resulting in a remaining block set S_{toc}. The blocks in S_{toc} are then sorted based on the following priority: (1) the identifier of the block creator, (2) the epoch number, and (3) the block height. Finally, the sorted sequence is appended to $\mathcal{V}_i$.

Fig. 3 illustrates the global sorting rule through an example. In the figure, C^0_1 represents the optimistic path, where blocks $B^{\langle 0,1\rangle}_1$, $B^{\langle 0,2\rangle}_1$, and $B^{\langle 0,3\rangle}_1$ are sequentially direct-committed. The direct commit of $B^{\langle 0,2\rangle}_1$ triggers the indirect commits of $B^{\langle 0,0\rangle}_2$, $B^{\langle 0,1\rangle}_2$, and $B^{\langle 0,0\rangle}_3$. These four blocks are written to $\mathcal{V}_i$ in the order $B^{\langle 0,2\rangle}_1$, $B^{\langle 0,0\rangle}_2$, $B^{\langle 0,1\rangle}_2$, $B^{\langle 0,0\rangle}_3$, as determined by the sorting rules defined in Algorithm 5. Similarly, the direct commit of $B^{\langle 0,3\rangle}_1$ triggers the indirect commit of $B^{\langle 0,1\rangle}_0$, which is appended to $\mathcal{V}_i$ in the order $B^{\langle 0,1\rangle}_0$, $B^{\langle 0,3\rangle}_1$.

E. Path switching

As shown in Lines 18-19 of Algorithm 4, when a node p_i detects that the number of uncommitted blocks in a chain exceeds λ ($\lambda \geq 3$), it triggers a path switching operation. To ensure safety, p_i suspends voting for blocks on the current optimistic path (Line 20). The path switching process consists of two parts: finalizing the previous path and initiating the next path.

1) Finalize the previous path: The node p_i first needs to align with other nodes on the number of committed blocks in the previous path, which is achieved by invoking the Π_{align} protocol (as shown in Line 21 of Algorithm 4). The construction of Π_{align} is presented in Algorithm 6, where each node first broadcasts the QC of its latest certified block from the previous path. Once $n - f$ QCs are collected, the node selects the one with the largest height and inputs this QC along with its corresponding height into the tcv^2-BA protocol Π_{tcvv}. Π_{tcvv} then returns a consistent block height to all nodes. Based on this height, nodes directly commit the previously uncommitted blocks, again by invoking the *GlobalSort* protocol (Lines 22-24 in Algorithm 4).

For the node that generates the previous optimistic path (i.e., p_{oid}), it concludes the chain associated with that path. This conclusion is accomplished by setting the $conc_v$ variable to `true` (Line 13), which is then passed to the $NewChain$ function in Algorithm 2. Furthermore, this node generates a new chain where the epoch number is incremented by 1 (Lines 14-15). Correspondingly, all nodes update the epoch number for p_{oid}'s latest chain (Line 16) and switch their optimistic path to the chain of the next node (Line 17).

2) Initiate the new path: Upon switching to the new optimistic path, uncommitted blocks on this path are first examined. If any blocks can be committed via the two-phase rule, they are immediately committed (as shown in Lines 9-11 of Algorithm 4). As a result, when multiple blocks are on the new path, these blocks, except the last two, can be committed without waiting, effectively reducing the latency after path switching.

F. Adaptive adjustment of λ

It is evident that the setting of the λ value may impact the performance of Icarus. On the one hand, to tolerate occasional network fluctuations, λ should be set to a larger value to prevent unnecessary switching of the optimistic path. On the other hand, when the current optimistic path fails, we desire a smaller λ value to facilitate a quicker transition to a new optimistic path. To reconcile these conflicting requirements, we design a dynamic λ adjustment mechanism composed of two parts: successive halving and probing recovery.

1) Successive halving: λ is dynamically adjusted within the predefined range $[\lambda_l, \lambda_h]$, where $\lambda_h = \lambda_l \cdot 2^k$. It is initialized to its maximum value, λ_h. We evaluate the lifespan of an optimistic path by monitoring its block generation. If no new blocks are generated on this path between its designation as the optimistic path and the subsequent path switch, the system is deemed to be under unfavorable conditions, and λ is halved. Conversely, a successful block generation leaves λ unchanged. This halving process continues until λ reaches its minimum value λ_l after k decrements.

2) Probing recovery: Once λ reaches its minimum value λ_l, a mechanism is needed to detect whether the system has recovered to a favorable condition. This is accomplished through a probing recovery procedure. First, we execute m optimistic paths using λ_l. Then, we tentatively restore λ to its maximum value λ_h for a single path, which we refer to as a trial path. If new blocks are successfully generated during the lifespan of this trial path, it indicates that the system has returned to a favorable condition, and the successive halving process restarts. Conversely, if this trial path fails to generate new blocks, the number of subsequent runs using λ_l is doubled to $2m$ before another trial path is attempted. This doubling repeats (to $4m$, $8m$, and so on) after each unsuccessful trial until one trial path successfully generates new blocks, at which point the successive halving process resumes.

IV. Correctness Analysis

In this section, we analyze the correctness of the Icarus protocol by verifying its adherence to the safety and liveness properties formally defined in Section II-B1. The proofs of these properties rely on several supporting lemmas. Due to space limitations, the proofs of some lemmas (namely, Lemmas 2, 3, and 5) are deferred to Appendix B.

A. Safety analysis

All blocks directly committed by a node p_i constitute a vector $\mathcal{D}_i$, which is append-only and shares structural similarities with $\mathcal{V}_i$. In accordance with the global sorting protocol described in Algorithm 5, the blocks within $\mathcal{V}_i$ can be partitioned into contiguous segments, each delineated by a directly committed block and encompassing all indirectly committed blocks it triggers. These segments are sequentially indexed as $Q_i[m]$, where $Q_i[m]$ specifically denotes the segment generated by the direct commit of $\mathcal{D}_i[m]$.

LEMMA 1. *On an optimistic path C, if the last block committed through the two-phase rule is $C[h]$, then all non-faulty nodes are guaranteed to commit blocks on C up to either $C[h]$ or $C[h+1]$.*

Proof. Without loss of generality, we assume node p_i commits block $C[h]$ through the two-phase rule, implying it has delivered blocks $C[h+1]$ and $C[h+2]$. Since $C[h+2]$ contains the QC for $C[h+1]$ (which consists of $n-f$ votes on $C[h+1]$), at least $n-2f$ non-faulty nodes have delivered $C[h+1]$ and possess the QC for $C[h]$.

On the other hand, since $C[h]$ is the last block committed via the two-phase rule, no node can deliver $C[h+k]$ (for any $k \geq 3$). This means no node can hold the QC for $C[h+l]$ (where $l \geq 2$).

Considering both points above, during the first round of Algorithm 6, at least $n-2f$ non-faulty nodes will broadcast the QC for either $C[h]$ or $C[h+1]$, while each node broadcasts QC for at most $C[h+m]$ (with $m \leq 1$). Given that $n-2f \geq f+1$, upon receiving $n-f$ `QC-EX` messages, each non-faulty node is guaranteed to receive the QC for $C[h]$ and may also receive the QC for $C[h+1]$.

According to Algorithm 6's design, each non-faulty node takes the height h or $h+1$ as the input parameter for Π_{tcvv} (Line 6). Leveraging the agreement and validity properties of tcv-BA, each non-faulty node ultimately outputs h or $h+1$ from Π_{tcvv} and commits blocks up to $C[h]$ or $C[h+1]$. □

LEMMA 2. *For any two non-faulty nodes p_i and p_j, if $\mathcal{D}_i[k] \neq \perp$ and $\mathcal{D}_j[k] \neq \perp$, then $\mathcal{D}_i[k] = \mathcal{D}_j[k]$*

LEMMA 3. *For any two non-faulty nodes p_i and p_j, if p_i and p_j commit the segments $Q_i[m]$ and $Q_j[m]$, respectively, then $Q_i[m] = Q_j[m]$.*

THEOREM 4 (SAFETY). *For any two non-faulty nodes p_i and p_j, if $\mathcal{V}_i[k] \neq \perp$ and $\mathcal{V}_j[k] \neq \perp$, then $\mathcal{V}_i[k] = \mathcal{V}_j[k]$.*

Proof. We assume blocks $\mathcal{V}_i[k]$ and $\mathcal{V}_j[k]$ reside within segments $Q_i[m_i]$ and $Q_j[m_j]$ respectively. Without loss of generality, let $m_i \leq m_j$.

By Lemma 3, for all $m < m_i$, we have $Q_i[m] = Q_j[m]$. This implies p_i and p_j commit identical numbers of blocks

across the first $m_i - 1$ segments, denoted by count t. Consequently, $\mathcal{V}_i[k]$ and $\mathcal{V}_j[k]$ must reside within the same segment, yielding $m_i = m_j$. Let $m_i = m_j = m$.

Furthermore, $\mathcal{V}_i[k]$ and $\mathcal{V}_j[k]$ correspond to the $(k-t)$-th blocks committed by p_i and p_j within the m-th segment, formally expressed as: $\mathcal{V}_i[k] = Q_i[m][k-t]$ and $\mathcal{V}_j[k] = Q_j[m][k-t]$.

Applying Lemma 3 again guarantees identical segments: $Q_i[m] = Q_j[m]$. Thus, we derive: $\mathcal{V}_i[k] = Q_i[m][k-t] = Q_j[m][k-t] = \mathcal{V}_j[k]$. □

B. Liveness analysis

1) Intuitional analysis: Our model in Section II-A assumes that messages originated by any non-faulty node are eventually delivered to all other non-faulty nodes. As a result, once a non-faulty node broadcasts a block, it is guaranteed to collect at least $n-f$ votes for that block, enabling the node to aggregate a QC and generate the next block. This key property ensures that the chain of every non-faulty node grows continuously.

Even if the adversary mounts a large-delay attack on each leader, after a finite duration, the chain of at least one non-faulty node will inevitably accumulate more than two uncommitted blocks. Similarly, if a temporary network partition occurs, the same conclusion holds once the partition heals. When this chain becomes the optimistic path, the two-phase committing rule described in Algorithm 4 ensures that at least one block on that chain can be committed, thereby upholding the liveness property.

2) Formal analysis: Next, we formally analyze the liveness of Icarus (as stated in Theorem 6), the proof of which relies on Lemma 5.

LEMMA 5. *In Icarus, blocks can be continuously committed.*

THEOREM 6 (LIVENESS). *If a transaction is added to the buffer of every non-faulty node, it will eventually be committed.*

Proof. Let `tx` denote the target transaction. Assuming `tx` has not been committed, all non-faulty nodes will incorporate `tx` into their generated blocks. Denote the set of such blocks as S. By Lemma 5, an unbounded number of blocks will get committed from this point forward. Since each committed block must reference at least $f+1$ newly generated blocks, it follows that a committed block must reference at least one block from S. This leads to the committing of `tx`. □

V. IMPLEMENTATION AND EVALUATION

A. Implementation

To validate the effectiveness of Icarus, we implement a prototype system and conduct comparative experiments against five baseline protocols: Ditto, ParBFT, sMVBA [19], FIN [20], and Tusk [24]. Ditto and ParBFT represent serial and parallel dual-path protocols, respectively, while sMVBA and FIN serve as two representative purely asynchronous protocols, and Tusk denotes a *Directed Acyclic Graph* (DAG)-based protocol. Ditto, ParBFT, sMVBA, and Tusk are evaluated using their open-source Rust implementations[1,2,3,4], which share a unified code framework. For fairness, Icarus is implemented within the same framework, and our implementation is made publicly available[5]. The framework employs a mempool-based transaction broadcasting mechanism to maximize bandwidth utilization. We implement the ABA protocol based on the MMR variant [27]. Specifically for FIN, due to the absence of an open-source Rust implementation, we develop it within the same framework.

Experiments are conducted on *Amazon Web Services* (AWS) to simulate a distributed environment, with nodes deployed across five global regions—N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1)—to ensure geographical distribution. Each node runs on an m5d.xlarge instance, equipped with 4 vCPUs, 16GB RAM, and up to 10Gbps bandwidth. To minimize experimental variance, each test is repeated three times with error bars included in the results. Each experiment runs for 5 minutes, and metrics are collected during a stable system status.

We focus on two key performance metrics: end-to-end latency and throughput. End-to-end latency is measured as the average time between transaction input and its successful commit, while throughput is quantified by the number of transactions committed per second. We consider two kinds of situations: favorable situations where all nodes operate normally without faults or artificial delay, enabling optimistic path execution for Ditto, ParBFT, and Icarus; and unfavorable situations where leader nodes are artificially delayed by 20 seconds to disrupt optimistic paths. Additionally, we assess performance under two more secure asynchronous conditions: temporary network partitions and random delays injected into each node. Due to space constraints, the experiments involving random delays are presented in Appendix C-B.

The transaction size is consistently fixed at 256 bytes, with each payload containing up to 1000 transactions and each block referencing a maximum of 32 payloads. For the parameter λ in Icarus, we employ two configuration strategies: a static setting and an adaptive adjustment mechanism. In the first four experimental studies (Sections V-B, V-C, V-D, and V-E), we utilize a static configuration to show that Icarus achieves better performance than existing alternatives even under a simple setup. The remaining experiments (Sections V-F and V-G) further illustrate Icarus's superior performance when the adaptive adjustment mechanism is enabled. Additionally, the timeout parameter in Ditto is set to 5 seconds.

B. Performance in favorable situations

To evaluate the performance of all protocols under favorable conditions, we deploy a system comprising seven nodes and set $\lambda = 10$ in Icarus. The appropriate value of λ is determined

[1]https://github.com/danielxiangzl/Ditto
[2]https://github.com/ac-dcz/parbft-parbft1-rust
[3]https://github.com/ac-dcz/sDumbo
[4]https://github.com/facebookresearch/narwhal
[5]https://github.com/CGCL-codes/icarus

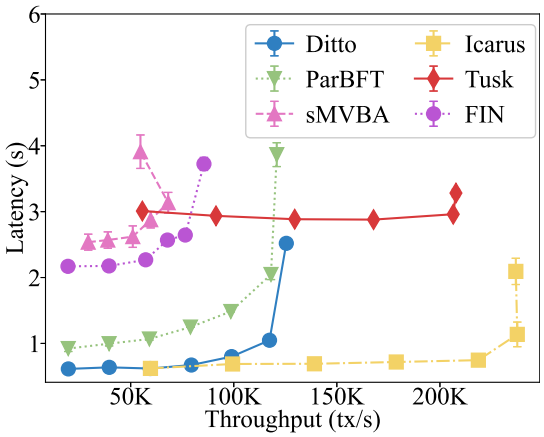


Fig. 4: Throughput *v.s.* latency in favorable situations

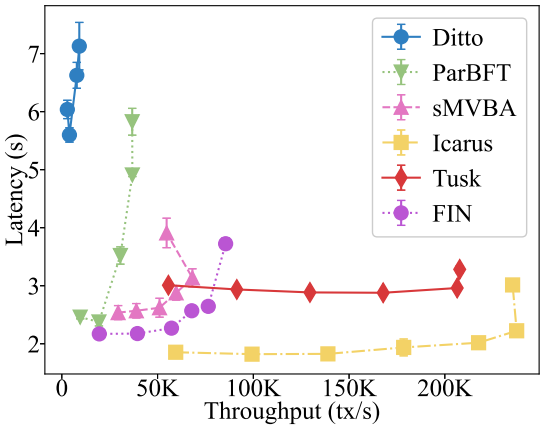


Fig. 5: Throughput *v.s.* latency in unfavorable situations

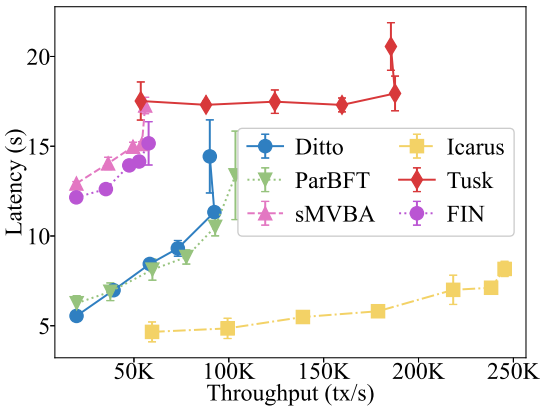


Fig. 6: Comparison in the presence of network partitions

through an empirical evaluation across a range of candidate values, the details of which are provided in Appendix C-A. In each experiment, we progressively increase the transaction input rate at the clients until the system reaches its saturation point. Throughout this process, we measure the resulting latency-throughput pairs, with the results presented in Fig. 4.

As shown in the figure, Icarus achieves low latency comparable to that of Ditto, since both protocols exploit an optimistic path for block committing. Additionally, Icarus reaches a peak throughput of 237.3K TPS, substantially outperforming Ditto (125.5K TPS), ParBFT (98.6K TPS), FIN (76.5K TPS), and sMVBA (68.2K TPS). This performance gain is attributed to Icarus's parallel-chain design, wherein each node independently broadcasts its own chain. By enabling blocks in other chains to be committed through the optimistic path, Icarus significantly enhances concurrency in block committing, thereby maximizing throughput.

On the other hand, owing to its DAG structure, which enables parallel block processing, Tusk achieves a high throughput of 206.5K TPS. Nevertheless, its throughput remains marginally lower than that of Icarus. This performance gap stems from the requirement in Tusk that each node must wait to deliver $n - f$ blocks from the previous round before it can generate a new block. This dependency constrains both the block production rate and the overall block committing speed. In contrast, Icarus allows each node to produce blocks without the need to wait for or reference $n-f$ blocks, thereby enabling more flexible and potentially faster block generation.

C. Performance in unfavorable situations

In the unfavorable situation, we similarly deployed seven nodes and set the parameter λ to 10. The experimental results are presented in Fig. 5.

By comparing the performance of different protocols, we observe that Icarus consistently achieves higher peak throughput than all other protocols, with particularly significant gains over Ditto, ParBFT, FIN, and sMVBA. Specifically, Icarus's peak throughput is approximately 1.2x that of Tusk, 3.1x that of FIN, 3.5x that of sMVBA, 12.2x that of ParBFT, and 30.4x that of Ditto. A comparison between Fig. 4 and Fig. 5 further reveals that due to the failure of the optimistic path, the peak throughput of both Ditto and ParBFT drops sharply, whereas Icarus maintains a high peak throughput. This resilience is attributed to Icarus's design: while the leader on the optimistic path may be delayed, other nodes continue to broadcast new blocks, allowing the system to sustain a high block production rate. Moreover, upon path switching, Icarus can leverage the new optimistic path to indirectly commit blocks in other chains, thereby maintaining high throughput.

In terms of latency, all of Icarus, Ditto, and ParBFT exhibit a noticeable increase under the unfavorable situation. This increase is due to the failure of the current optimistic path: Ditto and ParBFT must fall back to the pessimistic path, while Icarus must transition to a new optimistic path. However, Icarus still achieves lower latency than both Ditto and ParBFT. This is because once Icarus switches to a new optimistic path, if there are more than two uncommitted blocks on that path, it can immediately commit them, significantly accelerating block committing after the switch.

D. Performance under temporary network partitions

We maintain a configuration of seven nodes and set the λ value to 10. The network is alternately partitioned and then

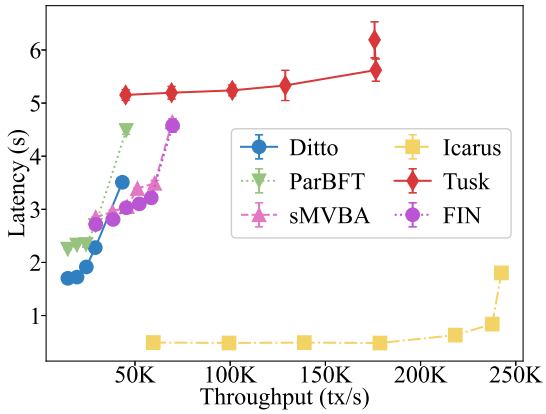
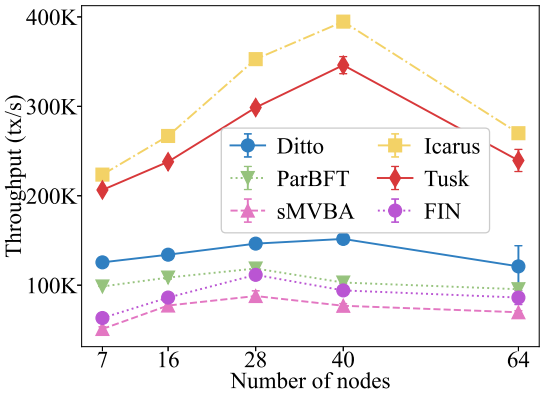
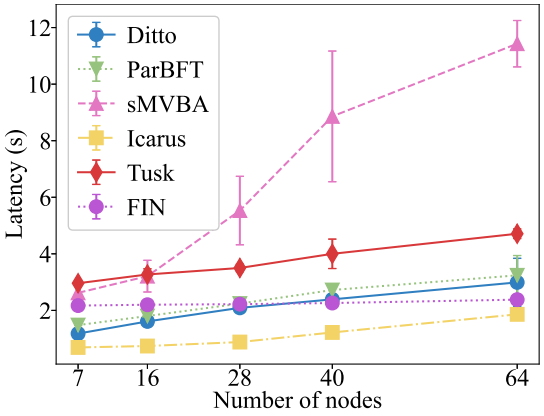


Fig. 7: Performance with a single rapid node



(a) Throughput comparison as the system scales



(b) Latency comparison as the system scales

Fig. 8: Scalability comparison under favorable situations

healed. Each partition lasts for 30 seconds, followed by a 60-second post-healing phase. When partitioned, the 7 nodes are split into two groups—one containing 3 nodes and the other 4 nodes—and network communication between the groups is blocked. As a result, no node can gather enough votes to form a QC and produce new blocks during the partition. The experimental results are shown in Fig. 6.

As observed in Fig. 6, Icarus maintains the highest peak throughput and the lowest latency among the compared protocols. A comparison between Fig. 6 and Fig. 4 reveals that, although temporary partitioning leads to a significant increase in latency for Icarus, its peak throughput remains largely unaffected. This can be attributed to the fact that each node continues to generate payloads during the partition, which are broadcast and referenced in blocks after network recovery, thus sustaining high throughput. Moreover, under temporary partitioning, Ditto exhibits higher latency than Icarus. This is because Ditto triggers a timeout and falls back to a pessimistic path for block committing after the partition heals, which introduces large latency. On the other hand, Icarus does not require a path switch upon recovery; it continues to commit blocks along the current path, resulting in lower latency.

E. Performance with a rapid node

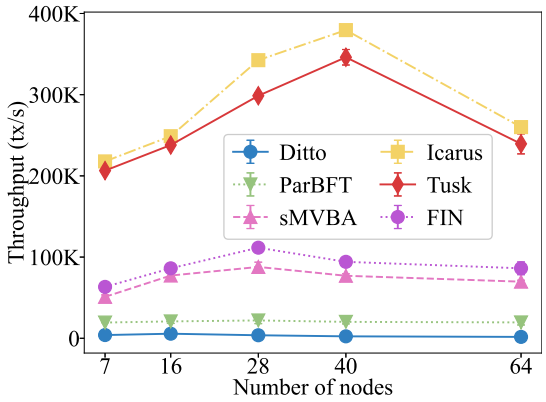
In Icarus, the switch of the optimistic path is triggered based on the number of uncommitted blocks on non-optimistic-path chains. When a particular chain exhibits a significantly higher block production rate than others, such that its blocks cannot be committed in time, Icarus initiates a switch of the optimistic path. This switching mechanism brings a key advantage: Icarus tends to migrate the optimistic path toward the fastest-growing chain. The experiments in this section are designed to validate this behavior.

We consider a system with seven nodes and set the λ parameter to 10. To simulate a scenario in which one node produces blocks at a significantly higher rate, we introduce an artificial delay of 500 ms to the block broadcasting step of all nodes except node p_2. The experimental results are shown in Fig. 7. As depicted, Icarus achieves significantly lower latency and higher throughput compared to the other four baseline protocols. Further log analysis reveals that Icarus quickly switches the optimistic path to the chain maintained by p_2 and remains on that chain, enabling fast and continuous block commitment.

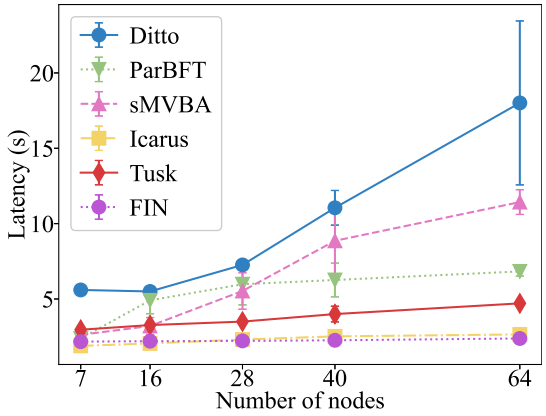
On the other hand, the 500 ms delay is not sufficient to trigger timeouts in Ditto, allowing it to continue committing blocks along its optimistic path. Similarly, for ParBFT, the optimistic path remains faster than the pessimistic fallback, leading ParBFT to commit blocks through the optimistic paths. However, under these conditions, the optimistic paths of both Ditto and ParBFT proceed at a much slower rate, resulting in higher latency and lower throughput compared to Icarus.

F. Scalability

We evaluate the scalability of the protocols by varying the number of nodes from 7 to 16, 28, 40, and 64, under both the favorable and unfavorable situations. These specific configurations are selected because they follow the $3f + 1$ form. In Icarus, the parameter λ is configured using the adaptive adjustment mechanism from Section III-F, with λ_l, λ_h, and m set to 5, 40, and 1, respectively. For each system scale, we gradually increase the transaction input rate to identify the system's saturation point, at which both latency and throughput are recorded.



(a) Throughput comparison as the system scales



(b) Latency comparison as the system scales

Fig. 9: Scalability comparison under unfavorable situations

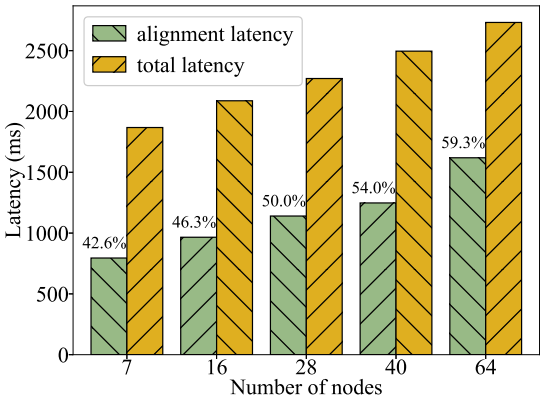


Fig. 10: Comparison between alignment and total latency

1) Scalability in favorable situations: The scalability results in favorable situations are shown in Fig. 8. It is evident that Icarus significantly outperforms its counterparts in both throughput and latency across all scales. In particular, when the system scales to 40 nodes, Icarus achieves a throughput approximately 2.5x that of Ditto, 3.7x that of ParBFT, 5.0x that of sMVBA, 4.1x that of FIN, and 1.1x that of Tusk.

As shown in Fig. 8a, all protocols exhibit a pattern where throughput initially increases with the number of nodes and then declines. This is because, in the early stage, adding more nodes contributes to higher transaction input rates, which in turn boosts throughput. However, as the system scale continues to grow, the overhead from broadcasting massive transactions starts to saturate the network bandwidth, leading to reduced consensus efficiency and a subsequent decline in throughput.

2) Scalability in unfavorable situations: In this set of experiments, we evaluate performance as the number of nodes increases, specifically when optimistic leaders are delayed by 20 seconds. The experimental results are presented in Fig. 9.

As shown in Fig. 9a, Icarus achieves the highest throughput across all system scales, with only marginal degradation compared to its performance under favorable situations. In contrast, both Ditto and ParBFT suffer significant throughput drops. This advantage of Icarus can be attributed to its design: even when the leader's block broadcast is temporarily delayed, the remaining nodes continue to generate and broadcast blocks, thereby sustaining high overall throughput.

From Fig. 9b, it can be seen that Icarus and FIN consistently achieve the lowest latency among the compared protocols. Icarus maintains low latency due to its ability to immediately commit blocks of the new optimistic path after a path switch, eliminating the need for additional waiting. FIN's low latency, on the other hand, benefits from the efficiency of its ABA variant (i.e., RABA), which can output in as little as one communication round [20]. In the future, we plan to integrate the RABA protocol into Icarus to further improve its performance.

G. Latency analysis of the height-alignment protocol

In this section, we investigate the impact of the alignment protocol on overall latency. Since path switching and height alignment mainly occur under unfavorable conditions, we adopt the same experimental setup as in Section V-F2 and measure the latency required for height alignment across different system scales. The height-alignment latency is defined as the average time taken by all nodes from triggering a path switch to its successful completion. The results are shown in Fig. 10, which compares the alignment latency and total latency under different system scales. The values annotated above the alignment latency bars indicate their proportion relative to the total latency.

Overall, height alignment accounts for approximately 40% to 60% of the total latency. Comparing results across different system scales, it can be observed that the proportion of latency attributed to height alignment increases with the number of nodes. This is because the height alignment process invokes the tcv-BA protocol, whose internal implementation relies on a threshold signature scheme to realize the common coins. The time required to generate a complete threshold signature increases significantly with the number of nodes. To improve protocol performance under unfavorable situations, future work may focus on optimizing the alignment protocol and the underlying tcv-BA protocol.

VI. Related Work

A. Synchronous BFT

Early BFT protocols, exemplified by the Byzantine Generals Problem [31], primarily adopted synchronous network assumptions. This choice greatly simplifies the design complexity by leveraging the predictability of message delivery within a predetermined time bound Δ [32], [33], [34]. Recent research efforts, such as Sync-HotStuff [7] and Pili [35], have attempted to optimize synchronous protocols, though their practical deployment remains limited due to inherent theoretical constraints. The fundamental limitation stems from synchronous protocols' stringent network assumptions, which require all messages between non-faulty nodes to arrive within Δ time units. This critical dependency creates a dilemma: setting Δ too small risks violating the synchronization assumption and compromising safety, while excessive Δ values significantly degrade protocol efficiency.

B. Partially synchronous BFT

As a network assumption that lies between synchronous and asynchronous models, the partially-synchronous network assumption offers a more realistic foundation compared to the synchronous model, while simultaneously providing greater efficiency than the asynchronous counterpart. Since its introduction by Dwork et al. [36], the partially-synchronous assumption has been widely adopted in the design of numerous BFT consensus protocols. Among these, the PBFT protocol proposed by Castro et al. [9] stands out as the most representative example. Building upon PBFT, researchers have proposed various optimization strategies, including the introduction of fast paths [37], the utilization of trusted hardware [38], and the adoption of more refined failure models [26].

The rise of blockchain technology has introduced new insights into the design of BFT protocols. By packaging transactions into blocks and connecting these blocks into a chain using block hashes—or more precisely, QC—the broadcast and voting processes for blocks are structured in a pipelined manner. This approach is epitomized by the HotStuff protocol [10]. On this basis, further optimizations to HotStuff have been proposed by various researchers, aiming to reduce consensus latency [21], [39] or enhance the success rate of block committing [22], [40].

Despite the long-standing attention and widespread adoption of partially-synchronous BFT, recent studies have cast doubt on its liveness properties [11], [12], [13]. Specifically, it has been shown that an adversary can manipulate the network in such a way that when a non-faulty node becomes the leader, blocks may fail to be delivered in a timely manner, whereas when a faulty node becomes the leader, it may refrain from broadcasting any blocks at all.

C. Asynchronous BFT

Asynchronous BFT protocols can be broadly categorized into two classes: asynchronous broadcast and asynchronous agreement [41], [42], [43], [44]. Asynchronous BFT broadcast protocols are designed to ensure the consistency of delivered data, with representative examples including *Reliable Broadcast* (RBC) [45] and *Consistent Broadcast* (CBC) [46]. However, when the broadcaster is faulty, it is possible that no node will deliver any data.

On the other hand, asynchronous agreement protocols aim to enable a group of nodes to reach consensus on a piece of data while guaranteeing that eventually every correct node will deliver the data. Depending on the form of the data being agreed upon, these protocols mainly fall into two categories: *Multi-Valued Byzantine Agreement* (MVBA) [18], [19], [47], [48] and *Asynchronous Binary Agreement* (ABA) [28], [38], [41]. MVBA can achieve agreement on any data that satisfies the external validity condition, while ABA is limited to binary values. As a result, MVBA can be directly employed as the consensus protocol underlying an SMR system [13], whereas ABA typically requires integration with underlying broadcast protocols such as RBC or CBC in order to realize SMR [11], [12], [49].

Asynchronous BFT protocols based on MVBA, in particular, make minimal assumptions about the underlying network, which endows them with strong robustness properties. However, this advantage comes at the cost of increased protocol complexity and reduced efficiency. In an effort to improve the performance of asynchronous BFT protocols, some studies have introduced optimistic paths and proposed dual-path protocols [14], [15], [16], [17]. Nevertheless, as analyzed in Section II-C, dual-path protocols suffer from several drawbacks, such as high protocol complexity, difficulty in timely path switching, and inefficient resource utilization. To address the issues inherent in dual-path protocols, we propose a single-path paradigm that achieves asynchronous BFT entirely through the use of optimistic paths, without relying on a backup pessimistic path.

In addition, some researchers have explored the incorporation of *Directed Acyclic Graph* (DAG) topologies into BFT protocol design, leveraging parallel block processing to enhance throughput [24], [29]. However, most DAG-based protocols require the election of a leader within each wave and rely on the leader to commit blocks, which significantly increases latency. Although protocols such as BullShark [50] and Wahoo [51] have introduced optimistic paths, they still depend on pessimistic paths to ensure liveness, thereby suffering from limitations similar to those of the previously discussed dual-path approaches. In fact, Icarus exhibits certain topological similarities with the aforementioned DAG-based protocols, as both allow a block to reference multiple other blocks. However, DAG-based protocols enforce that each block must reference at least $n-f$ other blocks, which inherently reduces the block generation rate and consequently limits throughput. In contrast, Icarus employs a more flexible topological structure by not mandating that each block references $n-f$ blocks, thereby achieving higher throughput.

VII. Conclusion

We present Icarus, a single-path asynchronous BFT protocol that exclusively employs optimistic paths. By eliminating the reliance on separate pessimistic paths, Icarus avoids the high overhead and implementation complexity inherent in the MVBA protocol used within the dual-path paradigm. Its rotating-chain mechanism and tcv^2-BA component jointly guarantee liveness and safety even under unfavorable situations. Furthermore, the parallel-chain structure and inter-chain references enable blocks from non-optimistic-path chains to be committed, thereby achieving high throughput. Formal analysis and experimental results demonstrate that Icarus is both correct and performant, offering a promising direction for future asynchronous BFT protocols.

Acknowledgment

This work is supported by National Science and Technology Major Project 2022ZD0115301.

References

[1] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[2] X. Wang, S. Duan, J. Clavin, and H. Zhang, "BFT in blockchains: From protocols to use cases," *ACM Computing Surveys*, vol. 54, no. 10, pp. 1–37, 2022.

[3] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proceedings of the 34th International Conference on Dependable Systems and Networks*. IEEE, 2004, pp. 575–584.

[4] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Computing Surveys*, vol. 54, no. 1, pp. 1–38, 2021.

[5] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[6] G. Zhang, F. Pan, Y. Mao, S. Tijanic, M. Dangana, S. Motepalli, S. Zhang, and H.-A. Jacobsen, "Reaching consensus in the Byzantine empire: A comprehensive review of BFT consensus algorithms," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–41, 2024.

[7] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync HotStuff: Simple and practical synchronous state machine replication," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 106–118.

[8] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system," *arXiv preprint arXiv:1805.04548*, 2018.

[9] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 1999, pp. 173–186.

[10] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.

[11] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.

[12] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous BFT made practical," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2028–2041.

[13] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020, pp. 803–818.

[14] Y. Lu, Z. Lu, and Q. Tang, "Bolt-Dumbo transformer: Asynchronous consensus as fast as the pipelined BFT," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 2159–2173.

[15] R. Gelashvili, L. Kokoris Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Proceedings of the 26th International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 296–315.

[16] X. Dai, B. Zhang, H. Jin, and L. Ren, "ParBFT: Faster asynchronous BFT consensus with a parallel optimistic path," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 504–518.

[17] E. Blum, J. Katz, J. Loss, K. Nayak, and S. Ochsenreither, "Abraxas: Throughput-efficient hybrid asynchronous consensus," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 519–533.

[18] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous Byzantine agreement," in *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 337–346.

[19] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding Dumbo: Pushing asynchronous BFT closer to practice," *Cryptology ePrint Archive*, 2022.

[20] S. Duan, X. Wang, and H. Zhang, "Fin: Practical signature-free asynchronous common subset in constant time," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 815–829.

[21] X. Sui, S. Duan, and H. Zhang, "Marlin: Two-phase BFT with linearity," in *Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2022, pp. 54–66.

[22] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-HotStuff: A fast and robust BFT protocol for blockchains," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 2478–2493, 2023.

[23] M. O. Rabin, "Randomized Byzantine generals," in *Proceedings of the Annual Symposium on Foundations of Computer Science*. IEEE, 1983, pp. 403–409.

[24] 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 17th European Conference on Computer Systems*. ACM, 2022, pp. 34–50.

[25] H. Xu, C. Zhang, X. Liu, Y. Lv, S. Gan, L. Zhu, and K. Li, "A fast and practical sector-based BFT consensus with sublinear communication complexity," *IEEE Transactions on Networking*, 2026.

[26] X. Dai, L. Huang, J. Xiao, Z. Zhang, X. Xie, and H. Jin, "Trebiz: Byzantine fault tolerance with Byzantine merchants," in *Proceedings of the 38th Annual Computer Security Applications Conference*. ACM, 2022, pp. 923–935.

[27] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages," in *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*. ACM, 2014, pp. 2–9.

[28] I. Abraham, N. Ben-David, and S. Yandamuri, "Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement," in *Proceedings of the 41st ACM Symposium on Principles of Distributed Computing*. ACM, 2022, pp. 381–391.

[29] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*. ACM, 2021, pp. 165–175.

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[31] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[32] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[33] D. Dolev and H. R. Strong, "Authenticated algorithms for Byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[34] P. Feldman and S. Micali, "Optimal algorithms for Byzantine agreement," in *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. ACM, 1988, pp. 148–161.

[35] T. H. Chan, R. Pass, and E. Shi, "Pili: An extremely simple synchronous blockchain," *Cryptology ePrint Archive*, 2018.

[36] 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.

[37] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2007, pp. 45–58.
[38] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 46–56, 2005.
[39] J. Niu, F. Gai, M. M. Jalalzai, and C. Feng, "On the performance of pipelined HotStuff," in *Proceedings of the 40th Annual IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.
[40] N. Giridharan, F. Suri-Payer, M. Ding, H. Howard, I. Abraham, and N. Crooks, "BeeGees: stayin'alive in chained BFT," in *Proceedings of the 42nd ACM Symposium on Principles of Distributed Computing*. ACM, 2023, pp. 233–243.
[41] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1983, pp. 27–30.
[42] R. Canetti and T. Rabin, "Fast asynchronous Byzantine agreement with optimal resilience," in *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. ACM, 1993, pp. 42–51.
[43] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, no. 4, pp. 824–840, 1985.
[44] M. Correia, N. F. Neves, and P. Veríssimo, "From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures," *The Computer Journal*, vol. 49, no. 1, pp. 82–96, 2006.
[45] G. Bracha, "Asynchronous Byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.
[46] D. Dolev, "The Byzantine generals strike again," *Journal of Algorithms*, vol. 3, no. 1, pp. 14–30, 1982.
[47] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Proceedings of the 21st Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.
[48] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-MVBA: Optimal multi-valued validated asynchronous Byzantine agreement, revisited," in *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing*. ACM, 2020, pp. 129–138.
[49] H. Zhang and S. Duan, "PACE: Fully parallelizable BFT from reproposable Byzantine agreement," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 3151–3164.
[50] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "BullShark: DAG BFT protocols made practical," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 2705–2718.
[51] X. Dai, Z. Zhang, Z. Guo, C. Ding, J. Xiao, X. Xie, R. Hao, and H. Jin, "Wahoo: A DAG-based BFT consensus with low latency and low communication overhead," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 7508–7522, 2024.

APPENDIX A
ILLUSTRATING AN EXECUTION OF ALGORITHMS

Our illustration begins with Algorithm 4, which serves as the main entry point of Icarus. Initially, all nodes start with a chain number of 0; consequently, all values in Γ are initialized to 0. The chain of node p_0 is designated as the first optimistic path, with the oid set to 0 (Algorithm 4, Line 3).

Furthermore, each node p_i invokes the $NewChain$ function from Algorithm 2 to generate its first chain (Algorithm 4, Line 4). The $NewChain$ function continuously generates and broadcasts new blocks as long as no conclusion signal is received (Algorithm 2, Lines 5-6). New block generation is accomplished by calling the $GenBlk$ function from Algorithm 1. All non-faulty nodes vote on received blocks (Algorithm 2, Lines 14-16). Upon collecting $n - f$ votes, the broadcaster assembles a QC, which is then used as a parameter to generate the subsequent block (Algorithm 2, Lines 17-18).

The system then proceeds to commit blocks using the optimistic path. For the current optimistic path C^{ν}_{oid}, blocks are committed via a two-phase rule provided that no path switch is triggered (Algorithm 4, Lines 6-11). Committing a block on the optimistic path also commits and sorts its referenced ancestor blocks, which is implemented by invoking the $GlobalSort$ function in Algorithm 5.

When a path switch is triggered, the broadcaster node for the current path must initiate a new chain, again by invoking the $NewChain$ function from Algorithm 2. Each node then executes the $\Pi^{\langle i,e\rangle}_{align}$ protocol from Algorithm 6 to align the committed block height of the current path (Algorithm 2, Lines 18-21). Internally, $\Pi^{\langle i,e\rangle}_{align}$ calls the Π^{id}_{tcvv} function from Algorithm 3 to reach agreement on two consecutive integer values. After receiving the aligned height from $\Pi^{\langle i,e\rangle}_{align}$, if a node possesses any uncommitted blocks, it commits them in sequence, again by invoking the $GlobalSort$ function in Algorithm 5.

APPENDIX B
PROOF OF LEMMAS

LEMMA 2. *For any two non-faulty nodes p_i and p_j, if $\mathcal{D}_i[k] \neq \perp$ and $\mathcal{D}_j[k] \neq \perp$, then $\mathcal{D}_i[k] = \mathcal{D}_j[k]$*

Proof. Since the optimistic path operates in a serial manner, all optimistic paths can be sequentially numbered as $P[x]$. Each directly committed block belongs to one of these optimistic paths. Without loss of generality, we assume $\mathcal{D}_i[k]$ and $\mathcal{D}_j[k]$ are committed on paths $P[x_i]$ and $P[x_j]$ respectively, where $x_i \leq x_j$.

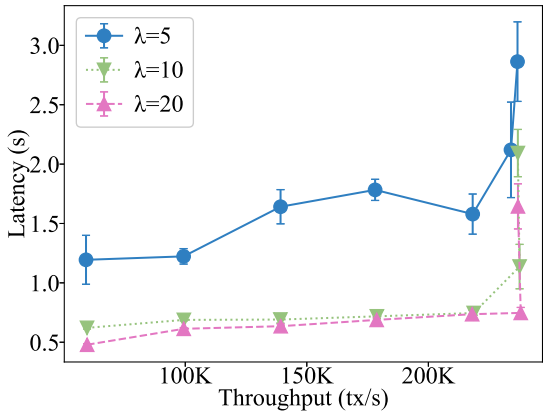
According to Lemma 1, for every path $P[x]$ ($x < x_i$), nodes p_i and p_j have committed an identical number of blocks. This necessitates $x_i = x_j$, thereby ensuring $\mathcal{D}_i[k]$ and $\mathcal{D}_j[k]$ are committed on the same optimistic path. For notational simplicity, we set $x_i = x_j = 0$ and denote the first path as C.

The proof proceeds by considering the following exhaustive cases:

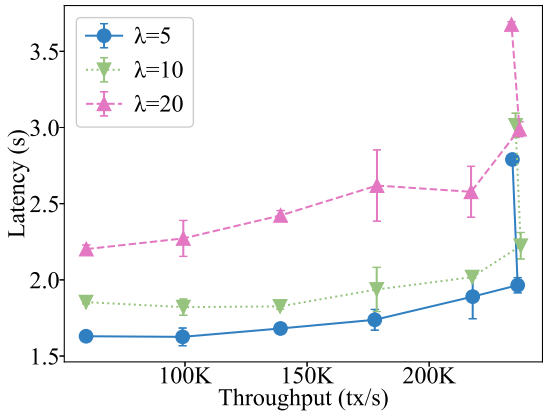
- Both blocks committed via two-phase rule: By the quorum mechanism, $\mathcal{D}_i[k] = \mathcal{D}_j[k]$.
- A single block committed via two-phase rule: Assume $\mathcal{D}_i[k]$ is committed via two-phase rule while $\mathcal{D}_j[k]$ is committed through path finalization. Since p_i commits block $C[h]$ at $\mathcal{D}_i[k]$, Lemma 1 dictates that p_j must have committed blocks up to $C[h]$ or $C[h+1]$ on C. Consequently, p_j necessarily commits $C[h]$ at $\mathcal{D}_j[k]$, ensuring $\mathcal{D}_i[k] = \mathcal{D}_j[k]$.
- Both blocks committed via path finalization: By the agreement property of tcv-BA, both p_i and p_j commit blocks up to $C[h]$ or $C[h+1]$ on C. In either scenario, p_i and p_j respectively commit $C[h]$ at $\mathcal{D}_i[k]$ and $\mathcal{D}_j[k]$, guaranteeing $\mathcal{D}_i[k] = \mathcal{D}_j[k]$.

This completes the proof of Lemma 2. □

LEMMA 3. *For any two non-faulty nodes p_i and p_j, if p_i and p_j commit the segments $Q_i[m]$ and $Q_j[m]$, respectively, then $Q_i[m] = Q_j[m]$.*



(a) Performance without delaying optimistic leaders



(b) Performance when delaying optimistic leaders

Fig. 11: Performance comparison using different λ values

Proof. Assume $Q_i[m]$ and $Q_j[m]$ are generated by the direct commits of $\mathcal{D}_i[m]$ and $\mathcal{D}_j[m]$, respectively. The proof proceeds via mathematical induction on the segment index m.

Base Case ($m = 0$): By Algorithm 5, the committed block sets satisfy $S_{com_i} = S_{com_j} = \emptyset$, where S_{com_i} and S_{com_j} denote the sets of blocks committed by p_i and p_j, respectively. From Lemma 2, we derive $\mathcal{D}_i[0] = \mathcal{D}_j[0]$, which immediately implies $S_{anc_i} = S_{anc_j}$ (ancestor block sets of $\mathcal{D}_i[0]$ and $\mathcal{D}_j[0]$). Consequently, the to-be-sorted sets satisfy $S_{toc_i} = S_{toc_j}$. Given identical sorting rules, the resulting sequences must satisfy $Q_i[0] = Q_j[0]$.

Inductive Step ($m \Rightarrow m+1$): Assume Lemma 3 holds for all $k \leq m$ (inductive hypothesis). For $k = m+1$:

1) By the inductive hypothesis, $Q_i[k] = Q_j[k]\ \forall k \leq m$.
2) During the direct commit of $\mathcal{D}_i[m+1]$ and $\mathcal{D}_j[m+1]$, S_{com_i} and S_{com_j} remain equivalent due to the inductive invariant.
3) Lemma 2 ensures $\mathcal{D}_i[m+1] = \mathcal{D}_j[m+1]$, yielding $S_{anc_i} = S_{anc_j}$.
4) The equality $S_{toc_i} = S_{toc_j}$ follows directly from the above, guaranteeing identical sorting outcomes: $Q_i[m+1] = Q_j[m+1]$.

Conclusion: By the principle of mathematical induction, Lemma 3 holds for all segment indices $m \geq 0$. □

LEMMA 5. *In Icarus, blocks can be continuously committed.*

Proof. We prove the lemma by contradiction. Assume no new blocks are committed in Icarus while the current optimistic path is the chain C. By protocol design, an optimistic path switch must inevitably occur under these conditions. According to Algorithm 4, this necessitates the existence of an alternative chain C' containing at least λ ($\lambda \geq 3$) uncommitted blocks. Two scenarios emerge: (1) If any block is committed before switching to C', it directly contradicts the assumption of zero new commits; (2) If no blocks are committed during the switch to C', C' will still retain $\lambda \geq 3$ uncommitted blocks, triggering to enforce new block committing—again violating the initial hypothesis. This contradiction proves that Icarus can continuously commit blocks. □

APPENDIX C
ADDITIONAL EVALUATION

A. Determining an appropriate value for static λ

In this section, we demonstrate the process of selecting an appropriate value of λ for a system of 7 nodes. Specifically, we examine λ values of 5, 10, and 20 under both favorable and unfavorable scenarios. The experimental results are presented in Fig. 11.

Fig. 11a illustrates Icarus's performance without introducing any delays to leaders for different λ values. It can be observed that when $\lambda = 5$, Icarus exhibits higher latency. This is due to the fact that smaller λ values increase the likelihood of premature or unnecessary path switches triggered by transient network fluctuations. When λ is increased to 10, such false triggers are significantly reduced, resulting in a substantial drop in latency. Further increasing λ to 20 yields no additional improvement in latency, as the probability of mis-triggered switches is already very low.

Fig. 11b shows the performance of Icarus when optimistic leaders are delayed by 20 seconds. When λ is set to 5, Icarus can quickly switch to a new optimistic path, thus achieving minimal latency. Conversely, augmenting the value of λ directly prolongs the path switching duration, which consequently results in larger latency. By comparing the cases of $\lambda = 10$ and $\lambda = 20$, we can find that the latency increase incurred by changing λ from 5 to 10 is substantially smaller than the corresponding increase measured between $\lambda = 10$ and $\lambda = 20$.

To sum up, when the system consists of seven nodes, setting $\lambda = 10$ yields a good empirical balance between the two situations.

In addition, we observe that the peak throughput of Icarus remains nearly constant across all λ settings. This is because Icarus continuously broadcasts blocks from all nodes, and variations in λ only affect the timing of block committing, not the number of blocks broadcast and committed per unit time.

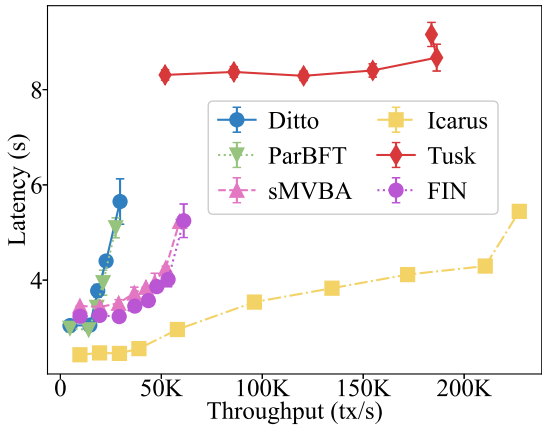


Fig. 12: Performance under per-node random delays

B. Evaluation under random delays on each node

In this set of experiments, we introduce random delays in the range of 500–1000 ms to the block-broadcasting process of each node. We continue to use a 7-node configuration and set $\lambda = 10$. The experimental results are presented in Fig. 12.

Comparing Fig. 4 and Fig. 12, we observe an obvious increase in latency across all protocols. Nevertheless, Icarus consistently achieves the lowest latency. Its baseline latency is approximately 81.1% of Ditto's, 83.7% of ParBFT's, 69.6% of sMVBA's, 75.7% of FIN's, and only 29.5% of Tusk's. The large increase in Tusk's latency arises from the fact that most of its blocks (i.e., non-leader blocks) require four to five rounds of block broadcasting, leading to significant cumulative delay. Furthermore, owing to the parallel block-broadcasting and commit mechanisms, both Tusk and Icarus maintain high throughput despite the injected delays.

APPENDIX D
ARTIFACT APPENDIX

This appendix outlines the evaluation methodology for our artifacts. In Section V, we present experimental results by deploying Icarus on *Amazon Web Service* (AWS) with replicas distributed across five geographically dispersed regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eunorth-1), and Tokyo (ap-northeast-1). Since configuring AWS involves a relatively complex process, we additionally provide local experimental instructions that can be executed on a single machine, thereby enabling easy validation of the code's functionality.

A. Description & Requirements

1) How to access: Source codes of Icarus are available on Github[6], with a permanent archival record at Zenodo[7]. Detailed configuration and step-by-step execution instructions are available in the README.md file in the repository.

2) Hardware dependencies: No special hardware requirements are required.

3) Software dependencies: Ubuntu 22.04 LTS is recommended as the operating system. Other Linux distributions can technically support deployment, as long as the operator can complete the configuration of the environment dependencies. The runtime environment mandates the installation of Python 3.9 or later, Rust (nightly version), Clang, and tmux terminal multiplexer for session management.

4) Benchmarks: We select Ditto, ParBFT, sMVBA, and Tusk as our benchmarks. Among these, Ditto[8] and ParBFT[9] represent serial and parallel dual-path protocols, respectively, while sMVBA[10] serves as a representative purely asynchronous protocol, and Tusk [11] denotes a Directed Acyclic Graph (DAG)-based protocol.

B. Artifact Installation & Configuration

Our repository can be downloaded using the `git clone https://github.com/DGJGzy/Icarus` command. We provide two ways to run our code: one is for local testing, and the other is for running on AWS.

1) Local testing: We offer two options for installing dependencies. The first is a 'dockerfile' that automatically generates a Docker image with all dependencies installed. The second is a manual installation method. To facilitate manual installation, we also provide a `build.sh` script. We recommend using the Docker approach for installation and execution. Detailed installation and configuration instructions will be provided in the *[Preparation]* part of Section D-D1.

2) Testing on AWS: After setting up AWS credentials and SSH keys, you can configure the environment by running commands such as `fab create` and `fab install`. For specific details, please refer to the *[Preparation]* part of Section D-D2.

C. Major Claims

- (C1): Icarus is a single-path asynchronous BFT protocol that exclusively leverages optimistic paths without relying on pessimistic paths.
- (C2): Under both favorable and unfavorable situations, Icarus consistently exhibits superior performance compared to other protocols, as shown in Sections V.B and V.C, as well as Figures 4 and 5. Furthermore, Icarus also demonstrates significantly better scalability, which is supported by the experimental results in Section V.F and Figure 8.
- (C3): In a situation where a single node (p_k) broadcasts blocks rapidly, Icarus can promptly switches the optimistic path to the chain led by p_k, thereby accelerating block committing and substantially surpassing the performance of baseline protocols, as demonstrated in Section V.E and Figure 7.

[6]https://github.com/DGJGzy/Icarus
[7]https://zenodo.org/records/17797381
[8]https://github.com/danielxiangzl/Ditto
[9]https://github.com/ac-dcz/parbft-parbft1-rust
[10]https://github.com/ac-dcz/sDumbo
[11]https://github.com/facebookresearch/narwhal

D. Evaluation

In this section, we present the workflows for running Icarus locally and on AWS.

1) Local experiment process: Local deployment of Icarus is relatively simple to implement.

[Preparation] There are two options to set up the testing environment.

Option 1: With Docker (Recommended). After changing to the project directory, execute the command below to build the Docker image, which has installed all dependencies required to run the experiment.

```
docker build -t icarus .
```

Then, execute the following command to launch a Docker container instance and enter its shell.

```
docker run -it --name icarus-dev icarus
/bin/bash
```

Option 2: Without Docker. You may choose to manually install the required dependencies, including:

- Rust
- Clang (dependency for RocksDB compilation)
- tmux (for running processes in the background)
- Python 3.9+

For convenience, we include a `build.sh` script that automates the installation of all required dependencies.

[Execution] After successfully launching the Docker container or completing the manual environment setup, you can now perform the following operations:

```
git clone https://github.com/DGJGzy/
Icarus
cd Icarus && cargo build
cd benchmark
pip install -r requirements.txt
```

These commands serve to clone the repository and install the required Python libraries. Note that the initial `cargo build` execution may take considerable time, as our implementation utilizes RocksDB—which requires compilation during this step.

To run the system, execute the `fab local` command within the Icarus/benchmark directory. The benchmark parameters can be customized in `fabfile.py`. Key configuration categories include:

[Benchmark parameters (bench_params)]

- `nodes`: number of replicas to run (default: 4)
- `duration`: test duration in seconds (default: 30)

[Node parameters (node_params)]

- `leader_delay`: whether to launch delay attacks on the leaders (default: False)
- `leader_delay_duration`: delay duration of the attacks on the leaders (default: 20)

[Results] When the `fab local` command completes, it displays an execution summary in the console and automatically saves detailed logs to the `logs` directory. You can use `fab logs` to parse these logs again, generating formatted results that match the console output and are saved to the `results` directory.

Using the default parameters described above, the Icarus system will run locally with four replicas deployed on a single machine. These replicas benefit from an optimized network environment, resulting in significantly reduced latency measurements. This differs from the results reported in Section V of our paper, which were obtained under *Wide Area Network* (WAN) conditions.

2) AWS-Based experiment process: The key difference between AWS and local deployments of Icarus is in the preparation phase.

[Preparation] To deploy Icarus on AWS, the following configuration steps must first be completed to set up the experimental environment.

- **Configure AWS credentials.** Enable programmatic access to your AWS account from your local machine. These credentials will authorize your system to programmatically create, modify, and delete EC2 instances.
- **Add SSH public key.** Manually add your SSH public key to each AWS region you intend to use.
- **Testbed Configuration** The file `settings.json` located in `Icarus/benchmark` contains all the configuration parameters of the testbed to deploy.
- **Testbed configuration.** Modify the `settings.json` file located in Icarus/benchmark to configure your testbed parameters.
- **Testbed deployment.** Execute `fab create` to provision new AWS instances. The creation logic is defined in `fabfile.py` under the 'create' task.
- **Dependency installation.** Run `fab install` to: (1) clone the repository on remote instances, and (2) install Rust language prerequisites.

For routine maintenance:

- Use `fab stop` to gracefully shut down the testbed.
- Use `fab start` to restart the testbed without recreating instances.

[Execution] After setting up the testbed, execute the protocol on AWS instances by running `fab remote`.

[Results] The `fab remote` command automatically collects logs from all replicas, enabling the result aggregation and log analysis similar to the local experiment workflow.

E. Customization

In addition to the parameters mentioned in Section D (`nodes`, `duration`, `ddoS`, `random_ddos`), you can also modify other parameters in `fabfile.py`, including:

- `tx_size`: transaction size in bytes (default: 256)
- `rate`: transactions input per second (default: 10,000)
- `faults`: Byzantine replicas to simulate (default: 0)
- `runs`: number of experimental runs (default: 1)