usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks

Lei Yang, Seo Jin Park, and Mohammad Alizadeh, *MIT CSAIL;*
Sreeram Kannan, *University of Washington;* David Tse, *Stanford University*

https://www.usenix.org/conference/nsdi22/presentation/yang

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks

Lei Yang[1], Seo Jin Park[1], Mohammad Alizadeh[1], Sreeram Kannan[2], David Tse[3]

[1]*MIT CSAIL* [2]*University of Washington* [3]*Stanford University*

Abstract

The success of blockchains has sparked interest in large-scale deployments of Byzantine fault tolerant (BFT) consensus protocols over wide area networks. A central feature of such networks is variable communication bandwidth across nodes and across time. We present DispersedLedger, an asynchronous BFT protocol that provides near-optimal throughput in the presence of such variable network bandwidth. The core idea of DispersedLedger is to enable nodes to propose, order, and agree on blocks of transactions without having to download their full content. By enabling nodes to agree on an ordered log of blocks, with a guarantee that each block is available within the network and unmalleable, DispersedLedger decouples bandwidth-intensive block downloads at different nodes, allowing each to make progress at its own pace. We build a full system prototype and evaluate it on real-world and emulated networks. Our results on a geo-distributed wide-area deployment across the Internet shows that DispersedLedger achieves 2× better throughput and 74% reduction in latency compared to HoneyBadger, the state-of-the-art asynchronous protocol.

1 Introduction

State machine replication (SMR) is a foundational task for building fault-tolerant distributed systems [25]. SMR enables a set of nodes to agree on and execute a replicated log of commands (or *transactions*). With the success of cryptocurrencies and blockchains, *Byzantine fault-tolerant SMR (BFT)* protocols, which tolerate arbitrary behavior from adversarial nodes, have attracted considerable interest in recent years [2, 5, 7, 15, 31, 39]. The deployment environment for these protocols differs greatly from standard SMR use cases. BFT implementations in blockchain applications must operate over wide-area networks (WAN), among possibly hundreds to thousands of nodes [2, 18, 31].

Large-scale WAN environments present new challenges for BFT protocols compared to traditional SMR deployments across a few nodes in datacenter. In particular, WANs are subject to *variability* in network bandwidth, both across different nodes and across time. While BFT protocols are secure in the presence of network variability, their performance can suffer greatly.

To understand the problem, let us consider the high-level structure of existing BFT protocols. BFT protocols operate in epochs, consisting of two distinct phases: (i) a *broadcast* phase, in which one or all of the nodes (depending on whether the protocol is leader-based [1, 39] or leaderless [17, 31]) broadcast a *block* (batch of transactions) to the others; (ii) an *agreement* phase, in which the nodes vote for blocks to append to the log, reaching a verifiable agreement (e.g., in the form of a quorum certificate [11]). From a communication standpoint, the broadcast phase is bandwidth-intensive while the agreement phase typically comprises of multiple rounds of short messages that do not require much bandwidth but are latency-sensitive.

Bandwidth variability hurts the performance of BFT protocols due to *stragglers*. In each epoch, the protocol cannot proceed until a super-majority of nodes have downloaded the blocks and voted in the agreement phase. Specifically, a BFT protocol on $N = 3f+1$ nodes (tolerant to f faults) requires votes from at least $2f + 1$ nodes to make progress [11]. Therefore, the throughput of the protocol is gated by the $(f+1)^{th}$ slowest node in each epoch. The implication is that low-bandwidth nodes (which take a long time to download blocks) hold up the high-bandwidth nodes, preventing them from utilizing their bandwidth efficiently. Stragglers plague even asynchronous BFT protocols [31], which aim to track actual network performance (without making timing assumptions), but still require a super-majority to download and vote for blocks in each epoch. We show that this lowers the throughput of these protocols well below the average capacity of the network on real WANs.

In this paper, we present DispersedLedger, a new approach to build BFT protocols that significantly improves performance in the presence of bandwidth variability. The key idea behind this approach is to decompose consensus into two steps, one of which is not bandwidth intensive and the other is. First, nodes agree on an ordered log of *commitments*, where each commitment is a small digest of a block (e.g., a Merkle root [30]). This step requires significantly less bandwidth than downloading full blocks. Later, each node downloads the blocks in the agreed-upon order and executes the transactions to update its state machine. The principal advantage of this approach is that each node can download blocks at its own pace. Importantly, slow nodes do not impede

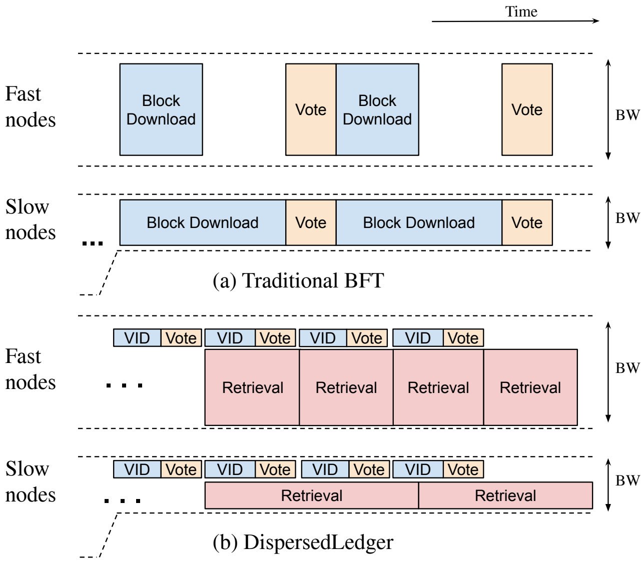


Figure 1: Impact of bandwidth variability on overall performance. Bcast: broadcast, Agmt: agreement. Fast nodes currently have a high bandwidth, while slow nodes currently have low bandwidth. (a) In traditional BFT protocols, the speed of consensus is always limited by the slow nodes since they take a long time to download the blocks. (b) DispersedLedger allows each node to download blocks at its own pace as permitted by its bandwidth.

the progress of fast nodes as long as they have a minimal amount of bandwidth needed to participate in the first step.

The key to realizing this idea is to guarantee the *data availability* of blocks. When a node accepts a commitment into the log, it must know that the block referred to by this commitment *is* available in the network and can be downloaded at a later time by any node in the network. Otherwise, an attacker can put a commitment of an unavailable block into the log, thus halting the system. To solve this problem, our proposal relies on *Verifiable Information Dispersal (VID)* [10]. VID uses erasure codes to store data across N nodes, such that it can be retrieved later despite Byzantine behavior. Prior BFT protocols like HoneyBadger [31] have used VID as a communication-efficient broadcast mechanism [10], but we use it to guarantee data availability. Specifically, unlike HoneyBadger, nodes in DispersedLedger do not wait to download blocks to vote for them. They vote as soon as they observe that a block has been *dispersed*, and the next epoch can begin immediately once there is agreement that dispersal has completed. This enables slow nodes to participate in the latest epoch, even if they fall behind on block downloads (retrieval). Such nodes can catch up on retrievals when their bandwidth improves. Figure 1 shows the structure of DispersedLedger, contrasting it to traditional BFT protocols.

Enabling nodes to participate in a consensus protocol with minimal bandwidth has applications beyond improving performance on temporally fluctuating bandwidth links. It also creates the possibility of a network with two types of nodes: high-bandwidth nodes and low-bandwidth nodes. All nodes participate in agreeing on the ordered log of commitments, but only the high-bandwidth nodes retrieve all blocks. Network participants can choose what mode to use at any time. For example, a node running on a mobile device can operate in the low-bandwidth mode when connected to a cellular network, and switch to high-bandwidth mode on WiFi to catch up on block retrievals. All nodes, both high-bandwidth and low-bandwidth, contribute to the network's security. Our approach is also a natural way to shard a blockchain [27], where different nodes only retrieve blocks in their own shard.

We make the following contributions:

- We propose a new asynchronous VID protocol, AVID-M (§3). Compared to the current state-of-the-art, AVID-M achieves 1–2 orders of magnitudes better communication cost when operating on small blocks (hundreds of KBs to several MBs) and clusters of more than a few servers.
- We design DispersedLedger (§4), an asynchronous BFT protocol based on HoneyBadger [31] with two major improvements: (i) It decomposes consensus into data availability agreement and block retrieval, allowing nodes to download blocks asynchronously and fully utilize their bandwidth (§4.2). (ii) It provides a new solution to the censorship problem [31] that has existed in such BFT protocols since [4] (§4.3). Unlike HoneyBadger, where up to f correct blocks can get dropped every epoch, our solution guarantees that *every* correct block is delivered (and executed). The technique is applicable to similarly-constructed protocols, and can improve throughput and achieve censorship resilience without advanced cryptography [31].
- We address several practical concerns (§4.5): (i) how to prevent block retrieval traffic from slowing down dispersal traffic, which could reduce system throughput; (ii) how to prevent constantly-slow nodes from falling arbitrarily behind the rest of the network; (iii) how to avoid invalid "spam" transactions, now that nodes may not always have the up-to-date system state to filter them out.
- We implement DispersedLedger in 8,000 lines of Go (§5) and evaluate it in multiple settings (§6), including two global testbeds on AWS and Vultr, and controlled network emulations. DispersedLedger achieves a throughput of 36 MB/s when running at 16 cities across the world, and a latency of 800 ms that is stable across a wide range of load. Compared to HoneyBadger, DispersedLedger has 105% higher throughput and 74% lower latency.

2 Background and Related Work

2.1 The BFT Problem

DispersedLedger solves the problem of Byzantine-fault-tolerant state machine replication (BFT) [25]. In general, BFT assumes a server-client model, where N servers maintain N replicas of a state machine. At most f servers are Byzantine and may behave arbitrarily. Clients may submit transactions to a correct server to update or read the state machine. A

BFT protocol must ensure that the state machine is replicated across all correct servers despite the existence of Byzantine servers. Usually, this is achieved by delivering a consistent, total-ordered log of transactions to all servers (nodes) [31]. Formally, a BFT protocol provides the following properties:

- **Agreement**: If a correct server executes a transaction m, then all correct servers eventually execute m.
- **Total Order**: If correct servers p and q both execute transactions m_1 and m_2, then p executes m_1 before m_2 if and only if q executes m_1 before m_2.
- **Validity**: If a correct client submits a transaction m to a correct server, then all correct servers eventually execute m.[1]

There are multiple trust models between BFT servers and the clients. In this paper, we assume a model used for *consortium blockchains* [2, 3, 6, 40], where servers and clients belong to organizations. Clients send their transactions through the servers hosted by their organization and trust these servers. Many emerging applications of BFT like supply chain tracing [14], medical data management [26], and cross-border transaction clearance [22] fall into this model.

2.2 Verifiable Information Dispersal

DispersedLedger relies on verifiable information dispersal (VID). VID resembles a distributed storage, where clients can *disperse* blocks (data files) across servers such that they are available for later *retrieval*. We provide a formal definition of VID in §3.1. The problem of *information dispersal* was first proposed in [37], where an erasure code was applied to efficiently store a block across N servers without duplicating it N times. [19] extended the idea to the BFT setting under the asynchrony network assumption. However, it did not consider Byzantine clients; these are malicious clients which try to cause two retrievals to return different blocks. *Verifiable* information dispersal (VID) was first proposed in [10], and solved this inconsistency problem. However, [10] requires that *every* node downloads the *full* block during dispersal, so it is no more efficient than broadcasting. The solution was later improved by AVID-FP [21], which requires each node to only download an $O(1/N)$ fraction of the dispersed data by utilizing fingerprinted cross-checksums [21]. However, because every message in AVID-FP is accompanied by the cross-checksum, the protocol provides low communication cost only when the dispersed data block is much larger than the cross-checksum (about $37N$ bytes). This makes AVID-FP unsuitable for small data blocks and clusters of more than a few nodes. In §3, we revisit this problem and propose AVID-M, a new asynchronous VID protocol that greatly reduces the per-message overhead: from $37N$ bytes to the size of a single hash (32 bytes), independent of the cluster size N, making the protocol efficient for small blocks and large clusters.

2.3 Asynchronous BFT protocols

A distributed algorithm has to make certain assumptions on the network it runs on. DispersedLedger makes the weakest assumption: *asynchrony* [28], where messages can be arbitrarily delayed but not dropped. A famous impossibility result [16] shows there cannot exist a deterministic BFT protocol under this assumption. With randomization, protocols can tolerate up to f Byzantine servers out of a total of $3f+1$ [24]. DispersedLedger achieves this bound.

Until recently [31], asynchronous BFT protocols have been costly for clusters of even moderate sizes because they have a communication cost of at least $O(N^2)$ [8]. HoneyBadger [31] is the first asynchronous BFT protocol to achieve $O(N)$ communication cost per bit of committed transaction (assuming batching of transactions). The main structure of HoneyBadger is inspired by [4], and it in turn inspires the design of other protocols including BEAT [15] and Aleph [17]. In these protocols, all N nodes broadcast their proposed blocks in each epoch, which triggers N parallel Binary Byzantine Agreement (BA) instances to agree on a subset of blocks to commit. [10] showed that VID can be used as an efficient construction of reliable broadcast, by invoking retrieval immediately after dispersal. HoneyBadger and subsequent protocols use this construction as a blackbox. BEAT [15] explores multiple tradeoffs in HoneyBadger and proposes a series of protocols based on the same structure. One protocol, BEAT3, also includes a VID subcomponent. However, BEAT3 is designed to achieve BFT *storage*, which resembles a distributed key-value store.

2.4 Security Model

Before proceeding, we summarize our security model. We make the following assumptions:

- The network is asynchronous (§2.3).
- The system consists of a fixed set of N nodes (servers). A subset of at most f nodes are Byzantine, and $N \geq 3f+1$. N and f are protocol parameters, and are public knowledge.
- Messages are authenticated using public key cryptography. The public keys are public knowledge.

3 AVID-M: An Efficient VID Protocol

3.1 Problem Statement

VID provides the following two primitives: `Disperse`(B), which a client invokes to disperse block B, and `Retrieve`, which a client invokes to retrieve block B. Clients invoke the `Disperse` and `Retrieve` primitives against a particular *instance* of VID, where each VID instance is in charge of dispersing a different block. Multiple instances of VID may run concurrently and independently. To distinguish between these instances, clients and servers tag all messages of each VID instance with a unique ID for that instance. For each instance of VID, each server triggers a `Complete` event to indicate that the dispersal has completed.

A VID protocol must provide the following properties [10] for each instance of VID:

[1] Some recent BFT protocols provide a weaker version of validity, which guarantees execution of a transaction m only after being sent to *all* correct servers. This is referred to by different names: “censorship resilience” in HoneyBadger, and “fairness” in [8, 9].

- **Termination**: If a correct client invokes Disperse(B) and no other client invokes Disperse on the same instance, then all correct servers eventually Complete the dispersal.
- **Agreement**: If some correct server has Completed the dispersal, then all correct servers eventually Complete the dispersal.
- **Availability**: If a correct server has Completed the dispersal, and a correct client invokes Retrieve, it eventually reconstructs some block B'.
- **Correctness**: If a correct server has Completed the dispersal, then correct clients always reconstruct the *same* block B' by invoking Retrieve. Also, if a correct client initiated the dispersal by invoking Disperse(B) and no other client invokes Disperse on the same instance, then $B = B'$.

3.2 Overview of AVID-M

At a high level, a VID protocol works by encoding the dispersed block using an erasure code and storing the encoded *chunks* across the servers. A server knows a dispersal has completed when it hears from enough peers that they have received their chunks. To retrieve a dispersed block, a client can query the servers to obtain the chunks and decode the block. Here, one key problem is verifying the correctness of encoding. Without verification, a malicious client may distribute *inconsistent* chunks that have more than one decoding result depending on which subset of chunks are used for decoding, violating the Correctness property. As mentioned in §2.2, AVID [10] and AVID-FP solve this problem by requiring servers to download the chunks or fingerprints of the chunks from all correct peers and examine them during dispersal. While this eliminates the possibility of inconsistent encoding, the extra data download required limits the scalability of these protocols.

More specifically, while AVID-FP [21] can achieve optimal communication complexity as the block size $|B|$ goes to infinity, its overhead for practical values of $|B|$ and N (number of servers) can be quite high. This is because every message in AVID-FP is accompanied by a fingerprinted cross-checksum [21], which is $N\lambda + (N-2f)\gamma$ in size. Here, λ, γ are security parameters, and we use $\lambda = 32$ bytes, $\gamma = 16$ bytes as suggested by [21]. The key factor that limits the scalability of AVID-FP is that the size of the cross-checksum is proportional to N. Combined with the fact that a node receives $O(N)$ messages during dispersal, the overhead caused by cross-checksum increases quadratically as N increases. Fig. 2 shows the impact of this overhead. At $N > 40$, $|B| = 100$ KB, every node needs to download more than the *full size* of the block being dispersed.

We develop a new VID protocol for the asynchronous network model, Asynchronous Verifiable Information Dispersal with Merkle-tree (AVID-M). AVID-M is based on one key observation: as long as clients can independently verify the encoding during *retrieval*, the servers do not need to do the verification during dispersal. In AVID-M, a client invoking Disperse(B) commits to the set of (possibly inconsistent) chunks using a short, constant-sized commitment H. Then the server-side protocol simply agrees on H and guarantees enough chunks that match H are stored by correct servers. This can be done by transmitting only H in the messages, compared to the $O(N)$-sized cross-checksums in AVID-FP. During retrieval, a client verifies that the block it decodes produces the same commitment H when re-encoded.

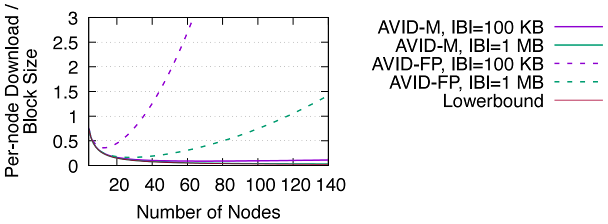


Figure 2: *Per-node* communication cost during dispersal of AVID-M and AVID-FP normalized over the size of the dispersed block. At $N = 128$ (the biggest cluster size in our evaluation), every node in AVID-M downloads as much as $1/32$ of a block, while a node in AVID-FP downloads $1.2\times$ the size of the full block.

Since AVID-M's per-message overhead is a small constant (32 bytes), it can scale to many nodes without requiring a large block size. In fact, AVID-M achieves a per-node communication cost of $O(|B|/N + \lambda N)$, much lower than AVID-FP's $O(|B|/N + \lambda N^2 + \gamma N^2)$. Fig. 2 compares AVID-M with AVID-FP. At $|B| = 1$ MB, AVID-M is close to the theoretical lowerbound[2] even at $N > 100$, while AVID-FP stops to provide any bandwidth saving (compared to every server downloading full blocks) after $N > 120$. Finally, we note that both AVID-M and AVID-FP rely on the security of the hash. So with the same hash size λ, AVID-M is no less secure than AVID-FP.

3.3 AVID-M Protocol

The **Dispersal algorithm** is formally defined in Fig. 3. A client initiates a dispersal by encoding the block B using an $(N-2f, N)$-erasure code and constructing a Merkle tree [30] out of the encoded chunks. The root r of the Merkle tree is a secure summary of the array of the chunks. The client sends one chunk to each server along with the Merkle root r and a Merkle *proof* that proves the chunk belongs to root r. Servers then need to make sure at least $N-2f$ chunks under the same Merkle root are stored at *correct* servers for retrieval. To do that, servers exchange a round of GotChunk(r) messages to announce the reception of the chunk under root r. When $N-f$ servers have announced GotChunk(r), they know at least $N-2f$ correct servers have got the chunk under the same root r, so they exchange a round of Ready(r) messages to collectively Complete the dispersal.

[2] Each node has to download at least $\frac{1}{N-2f}$-fraction of the dispersed data. This is to prevent a specific attack: a malicious client sends chunks to all f malicious servers plus $N-2f$ honest servers. For now the malicious servers do not deviate from the protocol, so the protocol must terminate (otherwise it loses liveness). Then the malicious servers do not release the chunks, so the original data must be constructed from the $N-2f$ chunks held by honest servers, so each honest server must receive an $\frac{1}{N-2f}$-fraction share.

```
Disperse($B$) invoker
1. Encode the input block $B$ using an $(N-2f,N)$-erasure
   code, which results in N chunks, $C_1,C_2,...,C_N$.
2. Form a Merkle tree with all chunks $C_1,C_2,...,C_N$, and
   calculate the Merkle tree root, $r$.
3. Send Chunk($r$,$C_i$,$P_i$) to the $i$-th server. Here $P_i$ is the
   Merkle proof showing $C_i$ is the $i$-th chunk under root $r$.
Disperse($B$) handler for the $i$-th server
• Upon receiving Chunk($r$,$C_i$,$P_i$) from a client:
    1. Check if $C_i$ is the $i$-th chunk under root $r$ by verifying
       the proof $P_i$. If not, ignore the message.
    2. Set MyChunk $= C_i$, MyProof $= P_i$, MyRoot $= r$ (all
       initially unset).
    3. Broadcast GotChunk($r$) if it has not sent a GotChunk
       message before.
• Upon receiving GotChunk($r$) from the $j$-th server:
    1. Increment ShareCount[$r$] (initially 0).
    2. If ShareCount[$r$] $\geq N-f$, broadcast Ready($r$).
• Upon receiving Ready($r$) from the $j$-th server:
    1. Increment ReadyCount[$r$] (inititally 0).
    2. If ReadyCount[$r$] $\geq f+1$, broadcast Ready($r$).
    3. If ReadyCount[$r$] $\geq 2f+1$, set ChunkRoot $= r$.
       Dispersal is Complete.
```

Figure 3: Algorithm for Disperse(B). Servers ignore duplicate messages (same sender and same type). When broadcasting, servers also send the message to themselves.

```
Retrieve invoker
• Broadcast RequestChunk to all servers.
• Upon getting ReturnChunk($r$, $C_i$, $P_i$) from the $i$-th
  server:
    1. Check if $C_i$ is the $i$-th chunk under root $r$ by verifying
       the proof $P_i$. If not, ignore the messsage.
    2. Store the chunk $C_i$ with the root $r$.
• Upon collecting $N-2f$ or more chunks with the same
  root $r$:
    1. Decode using any $N-2f$ chunks with root $r$ to get
       a block $B'$. Set ChunkRoot $= r$ (initially unset).
    2. Encode the block $B'$ using the same erasure code to
       get chunks ${C_1}',{C_2}',...,{C_N}'$.
    3. Compute the Merkle root $r'$ of $C'_1,C'_2,...,C'_N$.
    4. Check if $r' =$ ChunkRoot. If so, return $B'$. Otherwise,
       return string "BAD_UPLOADER".
Retrieve handler for the $i$-th server
• Upon receiving RequestChunk, respond with message
  ReturnChunk(ChunkRoot, MyChunk, MyProof) if
  MyRoot = ChunkRoot. Defer responding if dispersal is
  not Complete or any variable here is unset.
```

Figure 4: Algorithm for Retrieve. Clients ignore duplicate messages (same sender and same type).

The **Retrieval algorithm** is formally defined in Fig 4. A client begins retrieval by requesting chunks for the block from all servers. Servers respond by providing the chunk, the Merkle root r, and the Merkle proof proving that the chunk belongs to the tree with root r. Upon collecting $N-2f$ different chunks with the same root, the client can decode and obtain a block B'. However, the client must ensure that other retrieving clients also obtain B' no matter which subset of $N-2f$ chunks they use – letting clients perform this check is a key idea of AVID-M. To do that, the client re-encodes B', constructs a Merkle tree out of the resulting chunks, and verifies that the root is the same as r. If not, the client returns an error string as the retrieved content.

The AVID-M protocol described in this section provides the four properties mentioned in §3.1. We provide a proof sketch for each property, and point to Appendix B for complete proofs.

Termination (Theorem B.2). A correct client sends correctly encoded chunks to all servers with root r. The $N-f$ correct servers will broadcast GotChunk(r) upon getting their chunk. All correct servers will receive the $N-f$ GotChunk(r) and send out Ready(r), so all correct servers will receive at least $N-f$ Ready(r). Because $N-f > 2f+1$, all correct servers will Complete.

Agreement (Theorem B.4). A server Completes after receiving $2f+1$ Ready(r), of which $f+1$ must come from correct servers. So all correct servers will receive at least $f+1$ Ready(r). This will drive all of them to send Ready(r). Eventually every correct server will receive $N-f$ Ready(r), which is enough to Complete ($N-f > 2f+1$).

Availability (Theorem B.6). To retrieve, a client must collect $N-2f$ chunks with the *same* root. This requires that at least $N-2f$ correct servers have a chunk for the same root. Now suppose that a correct server Completes when receiving $2f+1$ Ready(r). When this happens, at least one correct server has sent Ready(r). We prove that this implies that at least $N-2f$ correct servers must have sent GotChunk(r) (Lemma B.1),i.e., they have received the chunk. Assume the contrary. Then there will be less than $N-f$ GotChunk(r). Now a correct server only sends Ready(r) if it either receives (i) at least $N-f$ GotChunk(r), or (ii) at least $f+1$ Ready(r). Neither is possible (see Lemma B.1).

All correct servers agree on the same root upon Complete by setting ChunkRoot to the same value (Lemma B.5). To see why, notice that each server will only send one GotChunk per instance. If correct servers Complete with 2 (or more) ChunkRoots, then at least $N-f$ servers must have sent GotChunk for each of these roots. But $2(N-f) > N+f$, hence at least one correct server must have sent GotChunk for two different roots, which is not possible.

Correctness (Theorem B.9). First, note that two correct clients finishing Retrieve will set ChunkRoot to be the same, i.e., they will decode from chunks under the same Merkle root r (Lemma B.5). However, we don't know if two different subsets of chunks under r would decode to the same block, because a malicious client could disperse arbitrary data as chunks. To ensure consistency of Retrieve across

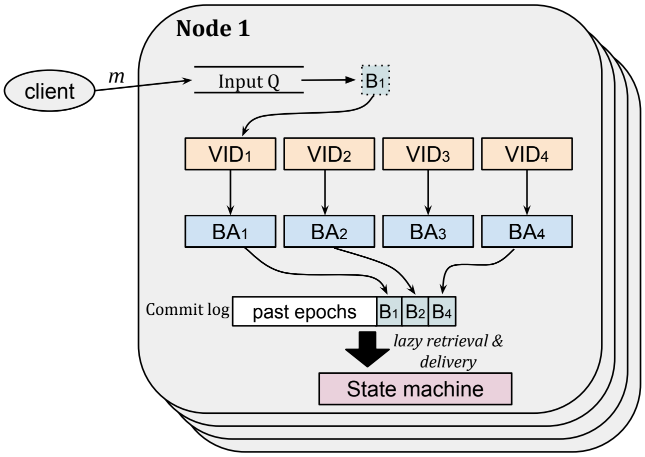


Figure 5: DispersedLedger architecture with $N = 4$. During this single epoch, 4 VIDs are initiated, one for each node, and three blocks B_1,B_2 and B_4 are committed.

different correct clients, every correct client re-encodes the decoded block B', calculates the Merkle root r' of the encoding result, and compares r' with the root r. There are two possibilities: (i) Some correct client gets $r' = r$. Then r corresponds to the chunks given by the *correct* encoding of B', so every correct client decoding from any subset of blocks under r will also get B' and $r' = r$. (ii) No correct client gets $r' = r$, i.e, all of them get $r' \neq r$. In this case, they all deliver the fixed error string. In either case, all correct clients return the same data (Lemma B.8).

4 DispersedLedger Design

4.1 Overview

DispersedLedger is a modification of HoneyBadger [31], a state-of-the-art asynchronous BFT protocol. HoneyBadger runs in epochs, where each epoch commits between $N - f$ to N blocks (at most 1 block from each node). As shown in Fig. 5, transactions submitted by clients are stored in each node's input queue. At the beginning of each epoch, every node creates a block from transactions in its input queue, and proposes it to be committed to the log in the current epoch. Once committed, all transactions in the block will eventually be retrieved and delivered to the state machine for execution.

DispersedLedger has two key differences with HoneyBadger. First, unlike HoneyBadger, a node in DispersedLedger does not broadcast its proposed block; instead, it *disperses* the proposed block among the entire cluster using AVID-M (which we will refer to as VID from here on). As shown in Fig. 5, there are N instances of VID in every epoch, one for each node. DispersedLedger then relies on N instances of Binary Agreement (BA, details below) [32] to reach a consensus on which proposed blocks have been successfully dispersed and thus should be committed in the current epoch. Once committed, a block can be retrieved by nodes lazily at any time (concurrently with future block proposals and dispersals). The asynchronous retrieval of blocks allows each node to adapt to temporal network bandwidth variations by adjusting the rate it retrieves blocks without slowing down other nodes.

```
Phase 1. Dispersal at the i-th server
1. Let $B_i^e$ be the block to disperse (propose) for epoch $e$.
2. Invoke Disperse($B_i^e$) on VID$_i^e$ (acting as a client).
• Upon Complete of VID$_j^e$ ($1 \leq j \leq N$), if we have not
  invoked Input on BA$_j^e$, invoke Input(1) on BA$_j^e$.
• Upon Output(1) of least $N - f$ BA instances, invoke
  Input(0) on all remaining BA instances on which we
  have not invoked Input.
• Upon Output of all BA instances,
    1. Let (local variable) $S_i^e \subset \{1...N\}$ be the indices of all
       BA instances that Output(1). That is, $j \in S_i^e$ if and
       only if BA$_j^e$ has Output(1) at the $i$-th server.
    2. Move to retrieval phase.
Phase 2. Retrieval
1. For all $j \in S_i^e$, invoke Retrieve on VID$_j^e$ to download
   full block $B_j^{e\prime}$.
2. Deliver $\{B_j^{e\prime} | j \in S_i^e\}$ (sorted by increasing indices).
```

Figure 6: Algorithm for single-epoch DispersedLedger.

In HoneyBadger, up to f correct blocks can be dropped in every epoch (§4.3). This wastes bandwidth and can lead to censorship where blocks from certain nodes are always dropped [31]. DispersedLedger's second innovation is a new method, called inter-node linking, that guarantees every correct block is committed.

DispersedLedger uses an existing BA protocol [32] that completes in $O(1)$ time (parallel rounds) with $O(N\lambda)$ per-node communication cost, where λ is the security parameter. In BA, each node provides a binary `Input({0,1})` as input to the protocol, and may get an `Output({0, 1})` event indicating the result of the BA instance. Formally, a BA protocol has the following properties:

- **Termination**: If all correct nodes invoke `Input`, then every correct node eventually gets an `Output`.
- **Agreement**: If any correct node gets `Output`(b) ($b \in \{0,1\}$), then every correct node eventually gets `Output`(b).
- **Validity**: If any correct node gets `Output`(b) ($b \in \{0,1\}$), then at least one correct node has invoked `Input`(b).

4.2 Single Epoch Protocol

In each epoch, the goal is to agree on a set of (the indices of) at least $N - f$ dispersed blocks which are available for later retrieval. An epoch contains N instances of VID and BA. Let VID$_i^e$ be the i-th ($1 \leq i \leq N$) VID instance of epoch e. VID$_i^e$ is reserved for the i-th node to disperse (propose) its block.[3] Let BA$_i^e$ be the i-th ($1 \leq i \leq N$) BA instance of epoch e. BA$_i^e$ is for agreeing on whether to commit the block dispersed by the i-th node.

[3] Correct nodes ignore attempts from another node j ($j \neq i$) to disperse into VID$_i^e$ by dropping `Chunk` messages for VID$_i^e$ from node j ($j \neq i$). Therefore, a Byzantine node cannot impersonate and disperse blocks on behalf of others.

Fig. 6 describes the single epoch protocol for the i-th node at epoch e. It begins by taking the block B_i^e to be proposed for this epoch, and dispersing it for epoch e through VID_i^e. Note that every block in the system is dispersed using a unique VID instance identified by its epoch number and proposing node.

Nodes now need to decide which blocks get committed in this epoch, and they should only commit blocks that have been successfully dispersed. Because there are potentially f Byzantine nodes, we cannot wait for all N instances of VID to complete because Byzantine nodes may never initiate their VID `Disperse`. On the other hand, nodes cannot simply wait for and commit the first $N-f$ VIDs to `Complete`, because VID instances may `Complete` in different orders at different nodes (hence correct nodes would not be guaranteed to commit the same set of blocks). DispersedLedger uses a strategy first proposed in [4]. Nodes use BA_i^e to explicitly agree on whether to commit B_i^e (which should be dispersed in VID_i^e). Correct nodes input 1 into BA_i^e only when VID_i^e `Completes`, so BA_i^e outputs 1 only if B_i^e *is* available for later retrieval. When $N-f$ BA instances have output 1, nodes give up on waiting for any more VID to `Complete`, and input 0 into the remaining BAs to explicitly signal the end of this epoch. This is guaranteed to happen because VID instances of the $N-f$ correct nodes will always `Complete` by the Termination property (§3.1). Once the set of committed blocks are determined, nodes can start retrieving the full blocks. After all blocks have been downloaded, a node sorts them by index number and delivers (executes) them in order.

The single-epoch DispersedLedger protocol is readily chained together epoch by epoch to achieve full SMR, as pictured in Fig. 5. At the beginning of every epoch, a node takes transactions from the head of the input buffer to form a block. After every epoch, a node checks if its block is committed. If not, it puts the transactions in the block back to the input buffer and proposes them in the next epoch. Also, a node delivers epoch e only after it has delivered all previous epochs.

4.3 Inter-node Linking

Motivation. An important limitation of the aforementioned single-epoch protocol (and all protocols with a similar construction [15, 31]) is that not all proposed blocks from correct nodes are committed in an epoch. An epoch only guarantees to commit $N-f$ proposed blocks, out of which $N-2f$ are guaranteed to come from correct nodes. In other words, at most f blocks proposed by correct nodes are dropped every epoch. Dropped blocks can happen *with or without* adversarial behavior. Transmitting such blocks wastes bandwidth, for example, reducing HoneyBadger's throughput by 25% in our experiments (§6.2). To make the matter worse, the adversary (if present) can determine which blocks to drop [31], i.e. at most f correct servers can be *censored* such that *no* block from these servers gets committed. HoneyBadger provides a partial mitigation by keeping the proposed blocks encrypted until they are committed so that the adversary cannot censor blocks by their content. The adversary *can*, however, censor blocks based on the proposing node.[4] This is unacceptable for consortium blockchains (§2.4), because the adversary could censor *all* transactions from certain (up to f) organizations. Moreover, HoneyBadger's mitigation relies on threshold cryptography, which incurs a high computational cost [15].

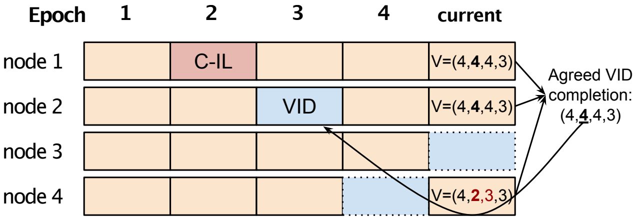


Figure 7: An example of commits by inter-node linking where $N=4, f=1$. Each box indicates a block proposed by a node at an epoch. Orange blocks are committed by BA. "VID" indicates that the block is dispersed but not committed. "C-IL" indicates a block committed by inter-node linking. Blue dotted boxes indicate a VID in progress. In the current epoch, after delivering the blocks from node 1, 2, and 4, the block proposed in epoch 3 by node 2 will be delivered by inter-node linking.

Our solution. We propose a novel solution to this problem, called *inter-node linking*, that guarantees *all* blocks from correct nodes are committed. Inter-node linking eliminates any censorship or bandwidth waste, and is readily applicable to similarly constructed protocols like HoneyBadger and BEAT. Notice that a block not committed by BA in a given epoch may still finish its VID. For example, in Fig. 7, the block proposed by node 2 in epoch 3 was dispersed but did not get selected by BA in that epoch. The core idea is to have nodes identify such blocks and deliver them in a consistent manner in later epochs.

Each node i keeps track of which VID instances have `Completed`, in the form of an array V_i^e of size N, which stores the local view at that node. When node i starts epoch e, it populates $V_i^e[j]$ (for all $1 \leq j \leq N$) with the largest epoch number such that all node j's VID instances up to epoch $V_i^e[j]$ have completed. For example, in Fig. 7, $(4,4,4,3)$ would be a valid array V for the current epoch, and would indicate that node 2's VID for epoch 3 has completed but node 4's VID in epoch 4 has not.

Each node i reports its local array V_i^e in the block B_i^e it proposes in each epoch (in addition to the normal block content). As shown in Fig. 7, the BA mechanism then commits at least $N-f$ blocks in each epoch. During retrieval for epoch e, a node first retrieves the blocks committed by BA in epoch e and delivers (executes) them as in the single-epoch protocol (§4.2). It then extracts the set of V arrays in the committed blocks, i.e. $\{V_j^e | j \in S_i^e\}$, and combines the information across these arrays to determine additional blocks that it should retrieve (and deliver) in this epoch. Note that $S_i^e = S_j^e$ for any two correct nodes i, j due to the Agreement property of BA, so all correct nodes

[4] HoneyBadger suggests sending transactions to all nodes to prevent censorship, but this isn't possible for consortium blockchains and still wastes bandwidth due to dropped blocks (§6.2).

will use the same set of observations and get the same result.[5]

Using the committed V arrays, the inter-node linking protocol computes an epoch number $E^e[j]$ for each node j. This is computed locally by each node i, but we omit the index i since all (correct) nodes compute the same value. Each node then retrieves and delivers (executes) all blocks from node j until epoch $E^e[j]$. To ensure total order, nodes sort the blocks, first by epoch number then by node index. They also keep track of blocks that have been delivered so that no block is delivered twice.

In computing $E^e[j]$, we must be careful to not get misled by Byzantine nodes who may report arbitrary data in their V arrays. For example, naively taking the largest value reported for node j across all V arrays, i.e., $\max_{k \in S_i^e} V_k^e[j]$, would allow a Byzantine node to fool others into attempting to retrieve blocks that do not exist. Instead, we take the $(f+1)^{\text{th}}$-largest value; this guarantees that at least one correct node i has reported in its array V_i^e that node j has completed all its VIDs up to epoch $E^e[j]$. Recall that by the Availability property of VID (§3.1), this ensures that these blocks are available for retrieval. Also, since all correct blocks eventually finish VID (Termination property), all of them will eventually be included in E^e and get delivered. We provide pseudocode for the full DispersedLedger protocol in Appendix C.

4.4 Correctness of DispersedLedger

We now analyze the correctness of the DispersedLedger protocol by showing it guarantees the three properties required for BFT (§2.1). Full proof is in Appendix D.

Agreement and Total Order (Theorem D.7). Transactions are embedded in blocks, so we only need to show Agreement and Total Order of *block* delivery at each correct node. Blocks may get committed and delivered through two mechanisms: BA and inter-node linking. First consider blocks committed by BA. BA's Agreement and VID's Correctness properties guarantee that (i) all correct nodes will retrieve the same set of blocks for each epoch, and (ii) they will download the same content for each block. Now consider the additional blocks committed by inter-node linking. As discussed in §4.3, correct nodes determine these blocks based on identical information (V arrays) included in the blocks delivered by BA. Hence they all retrieve and deliver the same set of blocks (Lemma D.2). Also, all correct nodes use the same sorting criteria (BA-delivered blocks sorted by node index, followed by inter-node-linked blocks sorted by epoch number and node index), so they deliver blocks in the same order.

Validity (Theorems D.5, D.6). Define "correct transactions" as ones submitted by correct clients to correct nodes (servers). We want to prove every correct transaction is eventually delivered (executed). This involves two parts: (i) correct nodes do not hang, so that every correct transaction eventually gets proposed in some correct block (Theorem D.5); (ii) all correct blocks eventually get delivered (Theorem D.6).

For part (i), note that all BAs eventually `Output`, since in every epoch at least $N-f$ BAs will `Output`(1) (Lemma D.3), and then all correct nodes will `Input`(0) to the remaining BAs and drive them to termination. Further, all blocks selected by BA or inter-node linking are guaranteed to be successfully dispersed, so `Retrieve` for them will eventually finish. By BA's Validity property, a BA only produces `Output`(1) when some correct node has `Input`(1), which can only happen if that node sees the corresponding VID `Complete`. Also, as explained in §4.3, inter-node linking only selects blocks that at least one correct node observes to have finished dispersal (Lemma D.4). By the Availability property of VID (§3.1), all these blocks are available for retrieval. For part (ii), note that all correct blocks eventually finish VID (Termination property). The inter-node linking protocol will therefore eventually identify all such blocks to have completed dispersal (Lemma D.4) and deliver them (if not already delivered by BA).

4.5 Practical Considerations

Running multiple epochs in parallel. In DispersedLedger, nodes perform dispersal sequentially, proceeding to the dispersal phase for the next epoch as soon as the dispersal for the current epoch has completed (all BA instances have `Output`). On the other hand, the retrieval phase of each epoch runs asynchronously at all nodes. To prevent slow nodes from stalling the progress of fast nodes, it is important that they participate in dispersal at as high a rate as possible, using only remaining bandwidth for retrieval. This effectively requires prioritizing dispersal traffic over retrieval traffic when there is a network bottleneck. Furthermore, a node can retrieve blocks from multiple epochs in parallel (e.g., to increase network utilization), but it must always deliver (execute) blocks in a serial order. Ideally, we want to fully utilize the network but prioritize traffic for earlier epochs over later epochs to minimize delivery latency. Mechanisms to enforce prioritization among different types of messages are implementation-specific (§5).

Constantly-slow nodes. Since DispersedLedger decouples the progress of fast and slow nodes, a natural question is: what if some nodes are constantly slow and do not have a chance to catch up? The possibility of some nodes constantly lagging behind is a common concern for BFT protocols. A BFT protocol cannot afford to wait for the slowest servers, because they could be Byzantine servers trying to stall the system [20]. Therefore the slow servers (specifically the f slowest servers) can be left behind, unable to catch up. Essentially, there is a tension between accommodating servers that are correct but slow, and preventing Byzantine nodes from influencing the system.

DispersedLedger expands this issue beyond the f slowest servers. We discuss two simple mitigations. First, the system designer could mandate a minimum average bandwidth per node such that all correct nodes can support the target system throughput over a certain timescale T. Every node

[5] If a particular `Retrieve` returns string "BAD_UPLOADER" or the block is ill formatted, we use array $[\infty, \infty, \ldots, \infty]$ as the observation.

must support the required bandwidth over time T but can experience lower bandwidth temporarily without stalling other nodes. Second, correct nodes could simply stop proposing blocks when too far behind, e.g., if their retrieval is more than P epochs behind the current epoch ($P = 1$ is the same as HoneyBadger). If enough nodes fall behind and stop proposing, it automatically slows down the system. A designer can choose parameters T or P to navigate the tradeoff between bandwidth variations impacting system throughput and how far behind nodes can get.

Spam transactions. In DispersedLedger, nodes do not check the validity of blocks they propose, deferring this check to the retrieval phase. This creates the possibility of malicious servers or clients spamming the system with invalid blocks.

Server-sent spam cannot be filtered even in conventional BFT protocols, because by the time other servers download the spam blocks, they have already wasted bandwidth. Similarly, HoneyBadger must perform BA (and incur its compute and bandwidth cost) regardless of the validity of the block, because by design, all BAs must eventually finish for the protocol to make progress [31]. Therefore, server-sent spam harms DispersedLedger and HoneyBadger equally. Fortunately, server-sent spam is bounded by the fraction of Byzantine servers (f/N).

On the other hand, client-sent spam is not a major concern in consortium blockchains (§2.1). In consortium blockchains, the organization is responsible for its clients, and a non-Byzantine organization would not spam the system.[6] For these reasons, some BFT protocols targeting consortium blockchains such as HyperLedger Fabric [2] forgo transaction filtering prior to broadcast for efficiency and privacy gains.

In more open settings, where clients are free to contact any server, spamming is a concern. A simple modification to the DispersedLedger protocol enables the same level of spam filtering as HoneyBadger. Correct nodes simply stop proposing new transactions when they are lagging behind in retrieval. Instead, they propose an empty block (with no transactions) to participate in the current epoch. In this way, correct nodes only propose transactions when they can verify them. Empty blocks still incur some overhead, so a natural question is: what is the performance impact of these empty blocks? Our results show that it is minor and this variant of DispersedLedger, which we call "DL-Coupled", retains most of the performance benefits (§6.2).

5 Implementation

We implement DispersedLedger in 8,000 lines of Go. The core protocol of DispersedLedger is modelled as 4 nested IO automata: `BA`, `VID`, `DLEpoch`, and `DL`. `BA` implements the binary agreement protocol proposed in [32]. `VID` implements our verifiable information dispersal protocol AVID-M described in §3.3. We use a pure-Go implementation of Reed-Solomon code [36] for encoding and decoding blocks, and an embedded key-value storage library [23] for storing blocks and chunks. `DLEpoch` nests N instances of `VID` and `BA` to implement one epoch of DispersedLedger (§4.2). Finally, `DL` nests multiple instances of `DLEpoch` and the inter-node linking logic (§4.3) to implement the full protocol.

Traffic prioritization. Prioritizing dispersal traffic over retrieval is made complicated because nodes cannot be certain of the bottleneck capacity for different messages and whether they share a common bottleneck. For example, rate-limiting the low-priority traffic may result in under-utilization of the network. Similarly, simply enforcing prioritization between each individual pair of nodes may lead to significant priority inversion if two pairs of nodes share the same bottleneck. In our implementation, we use a simple yet effective approach to achieve prioritization in a work conserving manner (without static rate limits) inspired by MulTcp [13]. For each pair of nodes, we establish two connections, and we modify the parameters of the congestion control algorithm of one connection so that it behaves like T ($T > 1$) connections . We then send high-priority traffic on this connection, and low-priority traffic on the other (unmodified) connection. At all bottlenecks, the less aggressive low-priority connection will back off more often and yield to the more aggressive high-priority connection. On average, a high-priority connection receives T times more bandwidth than a competing low-priority connection at the same bottleneck.[7] Note that in DispersedLedger, high-priority traffic consists of only a tiny fraction of the total traffic that a node handles ($1/20$ to $1/10$ in most cases as shown in §6.4), and its absolute bandwidth is low. Therefore our approach will not cause congestion to other applications competing at the same bottleneck. In our system, we set $T = 30$. We use QUIC as the underlying transport protocol and modify the `quic-go` [12] library to add the knob T for tuning the congestion control.

To prioritize retrieval traffic by epoch, we order retrieval traffic on a per-connection basis by using separate QUIC streams for different epochs. We modify the scheduler `quic-go` [12] to always send the stream with the lowest epoch number.

Rate control for block proposal. DispersedLedger requires some degree of batching to amortize the fixed cost of BA and VID. However, if unthrottled, nodes may propose blocks too often and the resulting blocks could be very small, causing low bandwidth efficiency. More importantly, since dispersal traffic is given high priority, the system may use up all the bandwidth proposing inefficient small blocks and leave no bandwidth for block retrieval. To solve this problem, our implementation employs a simple form of adaptive batching [29]. Specifically, we limit the block proposal rate using Nagle's algorithm [33]. A node only proposes a new block if

[6] A Byzantine organization could of course spam, but this is the same as the server-sent spamming scenario, in which DispersedLedger is no worse than HoneyBadger.

[7] Similar approaches have been used in other usecases to control bandwidth sharing among competing flows [34].

(i) a certain duration has passed since the last block was proposed, or (ii) a certain amount of data has accumulated to be proposed in the next block. In our implementation, we use 100 ms as the delay threshold, and 150 KB as the size threshold. This setup works well for all of our evaluation experiments.

6 Evaluation

Our evaluation answers the following questions:

1. What is the throughput and the latency of DispersedLedger in a realistic deployment?
2. Is DispersedLedger able to consistently achieve good throughput regardless of network variations?
3. How does the system scale to more nodes?

We compare DispersedLedger (DL) with the original HoneyBadger (HB) and our optimized version: HoneyBadger-Link. HoneyBadger-Link (HB-Link) combines the inter-node linking in DispersedLedger with HoneyBadger, so that every epoch, all (instead of $N-2f$) honest blocks are guaranteed to get into the ledger. We also experiment with DL-Coupled, a variant of DispersedLedger where nodes only propose new transactions when they are up-to-date with retrievals (§4.5).

6.1 Experimental Setup

We run our evaluation on AWS EC2. In our experiments, every node is hosted by an EC2 `c5d.4xlarge` instance with 16 CPU cores, 16 GB of RAM, 400 GB of NVMe SSD, and a 10 Gbps NIC. The nodes form a fully connected graph, i.e. there is a link between every pair of nodes. We run our experiments on two different scenarios. First, a *geo-distributed* scenario, where we launch VMs at 16 major cities across the globe, one at each city. We don't throttle the network. This scenario resembles the typical deployment of a consortium blockchain. In addition, we measure the throughput of the system on another testbed on Vultr (details are in Appendix A.2). Second, a *controlled* scenario, where we start VMs in one datacenter and apply artificial delay and bandwidth throttling at each node using Mahimahi [35]. Specifically, we add a one-way propagation delay of 100 ms between each pair of nodes to mimic the typical latency between distant major cities [38], and model the ingress and egress bandwidth variation of each node as independent Gauss-Markov processes (more details in §6.3). This controlled setup allows us to precisely define the variation of the network condition and enables fair, reproducible evaluations. Finally, to generate the workload for the system, we start a thread on each node that generates transactions in a Poisson arrival process.

6.2 Performance over the Internet

First, we measure the performance of DispersedLedger on our geo-distributed testbed and compare it with HoneyBadger.

Throughput. To measure the throughput, we generate a high load on each node to create an infinitely-backlogged system, and report the rate of confirmed transactions at each node. Because the internet bandwidth varies at different locations, we expect the measured throughput to vary as well. Fig. 8 shows the results. DispersedLedger achieves on average 105% better throughput than HoneyBadger. To confirm that our scheme is robust, we also run the experiment on another testbed using a low-cost cloud vendor. Results in §A.2 show that DispersedLedger significantly improves the throughput in that setting as well.

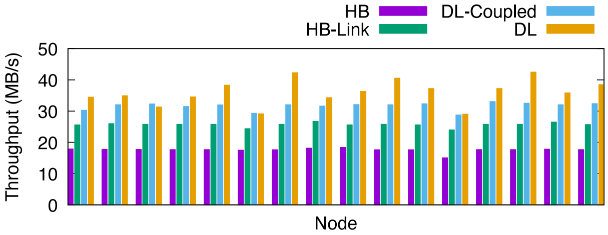
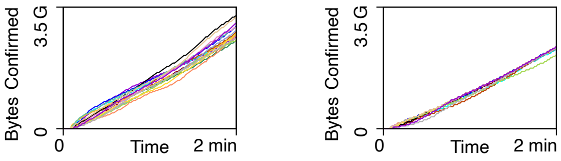


Figure 8: Throughput of each server running different protocols on the geo-distributed setting.



(a) DispersedLedger **(b)** HoneyBadger with linking

Figure 9: The amount of confirmed data over time when running DispersedLedger and HoneyBadger with inter-node linking on the geo-distributed testbed, plotted on the same scale. Each line represents one server.

DispersedLedger gets its throughput improvement mainly for two reasons. First, inter-node linking ensures all blocks that successfully finish VID get included in the ledger, so no bandwidth is wasted. In comparison, in every epoch of HoneyBadger at most f blocks may *not* get included in the final ledger. The bandwidth used to broadcast them is therefore wasted. As a result, inter-node linking provides at most a factor of $N/(N-f)$ improvement in effective throughput. To measure the gain in the real-world setting, we modify HoneyBadger to include the same inter-node linking technique and measure its throughput. Results in Fig. 8 show that enabling inter-node linking provides a 45% improvement in throughput on our geo-distributed testbed.

Second, confirmation throughput at different nodes are decoupled, so temporary slowdown at one site will not affect the whole system. Because the system is deployed across the WAN, there are many factors that could cause the confirmation throughput of a node to fluctuate: varying capacity at the network bottleneck, latency jitters, or even behavior of the congestion control algorithm. In HoneyBadger, the confirmation progress of all but the f slowest nodes are coupled, so at any time the whole system is only as fast as the $f+1$-slowest node. DispersedLedger does not have this limitation. Fig. 9 shows an example: DispersedLedger allows each node to always run at its own capacity. HoneyBadger couples the performance of most servers together, so all servers can only progress at the same, limited rate. In fact, notice that *every* node makes more progress with DispersedLedger compared to HoneyBadger (with linking) over the 2 minutes shown. The reason is that with HoneyBadger, different nodes become the straggler (the $(f+1)^{\text{th}}$-slowest node) at different time, stalling all other nodes. But with DispersedLedger, a slow node whose bandwidth improves can accelerate and make progress independently of others, making full use of time periods when it has high bandwidth. Fig. 8 shows that DispersedLedger achieves 41% better throughput compared to HoneyBadger with linking due to this improvement.

Finally, DL-Coupled is 12% slower than DL on average, but it still achieves 80% and 23% higher throughput on average than HoneyBadger and HoneyBadger with linking. Recall that DL-Coupled constrains nodes that can propose new transactions to prevent spamming attacks. The result shows that in open environments where spamming is a concern, DL-Coupled can still provide significant performance gains. In the rest of the evaluation, we focus on DL (without spam mitigation) to investigate our idea in its purest form.

Latency. Confirmation latency is defined as the elapsed time from a transaction entering the system to it being delivered. Similar to the throughput, the confirmation latency at different servers varies due to heterogeneity of the network condition. Further, for a particular node, we only calculate the latency of the transactions that this node *itself* generates, i.e. *local* transactions. This is a somewhat artificial metric, but it helps isolate the latency of each server in HoneyBadger and makes the results easier to understand. In HoneyBadger, a slow node only proposes a new epoch after it has confirmed the previous epoch, so the rate it proposes is coupled with the rate it confirms, i.e. it proposes 1 block after downloading $O(N)$ blocks. Due to this reason, an overloaded node does not have the capacity to even *propose* all the transactions it generates, and whatever transaction it proposes will be stale. When these stale transactions get confirmed at a fast node, the latency (especially the tail latency) at the fast nodes will suffer. Note that DispersedLedger does not have this problem, because all nodes, even overloaded ones, propose new transactions at a rate limited only by the egress bandwidth. In summary, choosing this metric is only advantageous to HoneyBadger, so the experiment remains fair. In Appendix §A.1, we provide further details and report the latency of all servers calculated for both local only, and all transactions.

We run the system at different loads and report the latency at each node. In Fig. 10, we focus on two datacenters: Mumbai, which has limited internet connection, and Ohio, which has good internet connection. We first look at the median latency. At low load, both HoneyBadger and DispersedLedger have similarly low median latency. But as we increase the load from 6 MB/s to 15 MB/s, the median latency of HoneyBadger increases almost linearly from around 800 ms to 3000 ms. This is because in HoneyBadger, proposing and confirming an epoch are done in lockstep. As the load increases, the proposed block becomes larger and takes longer to confirm. This in turn causes more transactions to be queued for the next block so the next proposed block remains large. Actually, the batch (all blocks in an epoch) size of HoneyBadger increases from 3.4 MB to 42.5 MB (200 KB to 2.5 MB per block) as we increase the load from 6 MB/s to 15 MB/s. Note that the block size is not chosen by us, but is naturally found by the system itself. In comparison, the latency of DispersedLedger only increases by a bit when the load increases, from 730 ms to 830 ms as we increase the load from 2 MB/s to 23 MB/s. The batch size ranges between 0.85 MB to 11.9 MB (50 KB to 700 KB per block).

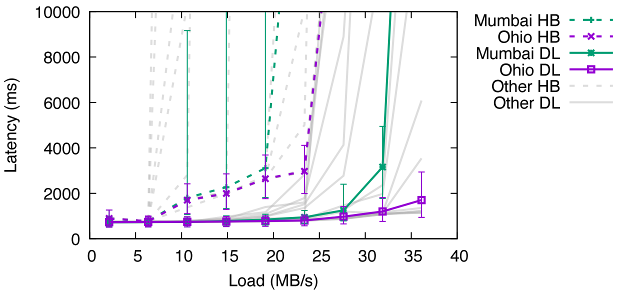


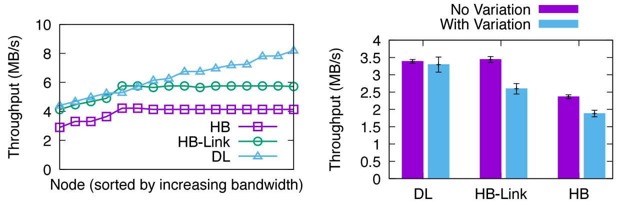
Figure 10: The median latency of DispersedLedger (solid) and HoneyBadger (dash) under different offered load. Error bar shows the 5-th and the 95-th percentiles. Two locations with good (Ohio) and limited (Mumbai) internet connection are highlighted.

We now look at the tail latency, which is important for service quality. At low load (6 MB/s), the 99-th percentile latency of DispersedLedger is 1000 ms across all servers, while that of HoneyBadger ranges from 1500 ms to 4500 ms. It suggests that DispersedLedger is more stable. As we increase the load, the tail (95-th percentile) latency of the Mumbai server immediately goes up. This is because HoneyBadger does not guarantee all honest blocks to be included in the ledger, and slow nodes are more likely to see their blocks being dropped from an epoch. When it happens, the node has to re-propose the same block in the next epoch, causing significant delay to the block. We note that the tail latency of the Ohio server goes up as well. In comparison, the tail latency of DispersedLedger at both Mumbai and Ohio stays low until very high load.

6.3 Controlled experiments

In this experiment, we run a series of tests in the controlled setting to verify if DispersedLedger achieves its design goal: achieving good throughput regardless of network variation. We start 16 servers in one datacenter, and add an artificial one-way propagation delay of 100 ms between each pair of servers to emulate the WAN latency. We then generate synthetic traces for each server that independently caps the ingress and egress bandwidth of the server. For each set of traces, we measure the throughput of DispersedLedger and HoneyBadger.

Spatial variation. This is the situation where the bandwidth varies across different nodes but stays the same over time. For the i-th node $(0 \le i < 16)$, we set its bandwidth to constantly be $10 + 0.5i$ MB/s. Fig. 11a shows that the throughput of



(a) Spatial variation **(b)** Temporal variation

Figure 11: Throughput of HoneyBadger (HB), HoneyBadger with linking (HB-Link), and DispersedLedger (DL) in the controlled experiments. Error bars in (b) show the standard deviation.

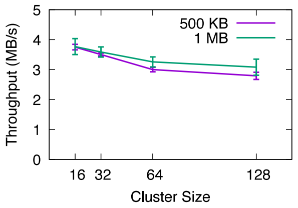


Figure 12: Throughput at different cluster size and block size. Error bars show the standard deviation.

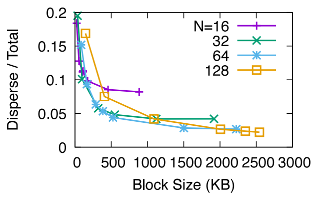


Figure 13: Fraction of dispersal traffic versus total traffic at different scale and block size.

HoneyBadger (with or without linking) is capped at the bandwidth of the fifth slowest server, and the bandwidth available at all faster servers are not utilized. In comparison, the throughput of DispersedLedger at different servers are fully decoupled. The achieved bandwidth is proportional to the available bandwidth at each server. DispersedLedger achieves this because it decouples block retrieval at different servers.

Temporal variation. We now look at the scenario where the bandwidth varies over time, and show that DispersedLedger is robust to network fluctuation. We model the bandwidth variation of each node as independent Gauss-Markov processes with mean b, variance σ, and correlation between consecutive samples α, and generate synthetic traces for each node by sampling from the process every 1 second. Specifically, we set $b = 10$ MB/s, $\sigma = 5$ MB/s, $\alpha = 0.98$ and generate a trace for each server, i.e. the bandwidth of each server varies independently but have the same distribution with mean bandwidth 10 MB/s. (We show an example of such trace in §A.3.) As a comparison, we also run an experiment when the bandwidth at each server does not fluctuate and stays at 10 MB/s. In our implementation (for all protocols), a node notifies others when it has decoded a block to stop sending more chunks. This optimization is less effective when all nodes have exactly the same fixed bandwidth because all chunks for a block will arrive at roughly the same time. So in this particular experiment, we disable this optimization to enable an apple-to-apple comparison of the fixed and variable bandwidth scenarios. Fig. 11b shows that as we introduce temporal variation of the network bandwidth, the throughput of DispersedLedger stays the same. This confirms that DispersedLedger is robust to network fluctuation. Meanwhile, the throughput of HoneyBadger and HoneyBadger with linking dropped by 20% and 25% respectively.

6.4 Scalability

In this experiment, we evaluate how DispersedLedger scales to a large number of servers. As with many evaluations of BFT protocols [31,39], we use cluster sizes ranging from 16 to 128.

Throughput. We first measure the system throughput at different cluster size N. For this experiment, we start all the servers in the same datacenter with a 100 ms one-way propagation delay on each link and a 10 MB/s bandwidth cap on each server. We also fix the block size to 500 KB and 1 MB. Fig. 12 shows that the system throughput slightly drops when N grows 8 times bigger from 16 nodes to 128 nodes. This is because the BA in the dispersal phase has a per-node cost of $O(N^2)$. With a constant block size, the messaging overhead takes a larger fraction as N increases. We notice that increasing the block size helps amortize the cost of VID and BA, and results in better system throughput.

Traffic for block dispersal. A metric core to the design of DispersedLedger is the amount of data a node has to download in order to participate in block dispersal, i.e. dispersal traffic. More precisely, we are interested in the *ratio* of dispersal traffic to the total traffic (dispersal plus retrieval). The lower this ratio, the easier it is for slow nodes to keep up with block dispersal, and the better DispersedLedger achieves its design goal. Fig. 13 shows this ratio at different scales and block sizes. First, we observe that increasing the block size brings down the fraction of dispersal traffic. This is because a large block size amortizes the fixed cost in VID and BA. Meanwhile, increasing the cluster size reduces the lower bound on the fraction of dispersal traffic. This is because in the VID phase, every node is responsible for an $1/(N-2f)$ slice of each block, and increasing N brings down this fraction.

7 Conclusion

We presented DispersedLedger, a new asynchronous BFT protocol that provides near-optimal throughput under fluctuating network bandwidth. DispersedLedger is based on a novel restructuring of BFT protocols that decouples agreement from the bandwidth-intensive task of downloading blocks. We implement a full system prototype and evaluate DispersedLedger on two testbeds across the real internet and a controlled setting with emulated network conditions. Our results on a wide-area deployment across 16 major cities show that DispersedLedger achieves 2× better throughput and 74% lower latency compared to HoneyBadger. Our approach could be applicable to other BFT protocols, and enables new applications where resilience to poor network condition is vital.

Acknowledgments

We would like to thank the National Science Foundation grants CNS-1751009 and CNS-1910676, the Cisco Research Center Award, the Microsoft Faculty Fellowship, and the Fintech@CSAIL program for their support.

References

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy*, pages 106–118. IEEE, 2020.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018.

[3] M. Belotti, N. Božić, G. Pujolle, and S. Secci. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

[4] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192. ACM, 1994.

[5] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[6] V. Buterin. On public and private blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/. Accessed: 2021-08-15.

[7] V. Buterin and V. Griffith. Casper the friendly finality gadget. *arXiv:1710.09437v4*, 2019.

[8] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology — CRYPTO 2001*, pages 524–541. Springer, 2001.

[9] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *2002 International Conference on Dependable Systems and Networks*, pages 167–176. IEEE, 2002.

[10] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems*, pages 191–201. IEEE, 2005.

[11] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[12] L. Clemente. quic-go: A QUIC implementation in pure Go. https://github.com/lucas-clemente/quic-go.

[13] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer Communication Review*, 28(3):53–69, 1998.

[14] M. Du, Q. Chen, J. Xiao, H. Yang, and X. Ma. Supply chain finance innovation using blockchain. *IEEE Transactions on Engineering Management*, 67(4):1045–1058, 2020.

[15] S. Duan, M. K. Reiter, and H. Zhang. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041. ACM, 2018.

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

[17] A. Gągol and M. Świętek. Aleph: A leaderless, asynchronous, Byzantine fault tolerant consensus protocol. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228. ACM, 2019.

[18] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[19] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *2004 International Conference on Dependable Systems and Networks*, pages 135–144. IEEE, 2004.

[20] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 568–580. IEEE, 2019.

[21] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 139–146. ACM, 2007.

[22] N. Kabra, P. Bhattacharya, S. Tanwar, and S. Tyagi. Mudrachain: Blockchain-based framework for automated cheque clearance in financial institutions. *Future Generation Computer Systems*, 102:574–587, 2020.

[23] A. Krylysov. pogreb: Embedded key-value store for read-heavy workloads written in Go. https://github.com/akrylysov/pogreb.

[24] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In *Automata, Languages and Programming*, pages 204–215. Springer, 2005.

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. In *Concurrency: The Works of Leslie Lamport*, pages 203–226. ACM, 2019.

[26] J. Liu, T. Liang, R. Sun, X. Du, and M. Guizani. A privacy-preserving medical data sharing scheme based on consortium blockchain. In *2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[27] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[28] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[29] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *2010 USENIX Annual Technical Conference*. USENIX Association, 2010.

[30] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378. Springer, 1987.

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

[32] A. Mostéfaoui, M. Hamouma, and M. Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 2–9. ACM, 2014.

[33] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, RFC Editor, 1984.

[34] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh. End-to-end transport for video QoE fairness. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, pages 408–423. ACM, 2019.

[35] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference*, pages 417–429. USENIX Association, 2015.

[36] K. Post. reedsolomon: Reed-Solomon erasure coding in Go. https://github.com/klauspost/reedsolomon.

[37] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[38] WonderNetwork. Global ping statistics. https://wondernetwork.com/pings/. Accessed: 2021-08-15.

[39] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.

[40] R. Zhang, R. Xue, and L. Liu. Security and privacy on blockchain. *ACM Computing Surveys*, 52(3), 2019.

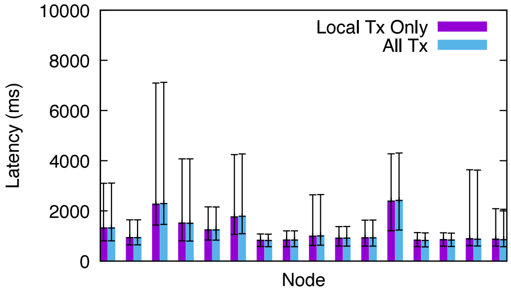
A Supplements to the Evaluations

A.1 Latency metric

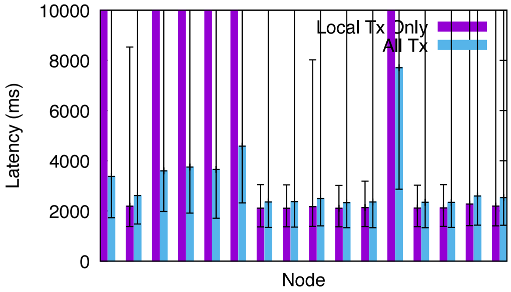
Here we justify counting only local transactions when calculating the confirmation latency. As mentioned in §6.2, we choose this metric to prevent overloaded servers from impacting the latency (especially the tail latency) of non-overloaded servers. Fig. 14 shows the latency of DispersedLedger and HoneyBadger under two metrics: counting all transactions, and counting only local transactions. Each system is running near its capacity. We observe that the latency (both the median and the tail) of DispersedLedger is the same under the two metrics, so choosing to count only local transactions in no way helps our protocol. For HoneyBadger, we observe that by counting all transactions, the median latency of the overloaded servers decreased. This is because the overloaded servers cannot get their local transactions into the ledger (so the local transactions have high latency), but can confirm *some* transactions from other non-overloaded servers. The median latency mostly represents these non-local transactions. Still, these servers are overloaded, and the latency numbers are meaningless because they will increase as system runs for longer. So the latency metric does not matter for the overloaded servers. Meanwhile, we observe that the tail latency of HoneyBadger on non-overloaded servers worsens a lot as we switch to counting all transactions. This is due to the transactions proposed by the overloaded nodes, and is the main reason that we choose to count only local transactions. In summary, counting only local transactions for latency calculation does not improve the latency of DispersedLedger, but helps improve the tail latency of non-overloaded servers in HoneyBadger, so choosing this metric is fair.

A.2 Throughput on another testbed over the internet

To further confirm that DispersedLedger improves the throughput of BFT protocols when running over the internet, we build another testbed on a low-cost cloud provider called Vultr. We use the $80/mo plan with 6 CPU cores, 16 GB of RAM, 320 GB of SSD, and an 1 Gbps NIC. At the moment of the experiment, Vultr has 15 locations across the globe, and



(a) DispersedLedger



(b) HoneyBadger

Figure 14: Confirmation latency of DispersedLedger and HoneyBadger when counting all transactions (All Tx) or only local transactions. Each system runs near its capacity (14.8 MB/s for HoneyBadger and 23.4 MB/s for DispersedLedger). The error bar shows the 5-th and 95-th percentiles.

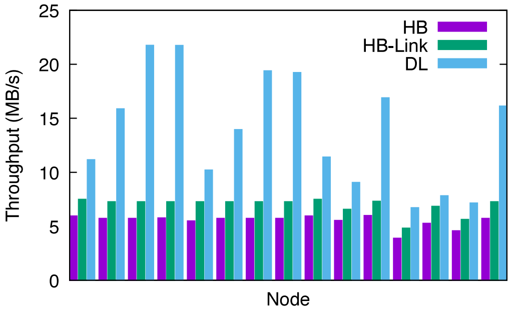


Figure 15: Throughput of each server running different protocols on the Vultr testbed. HB stands for HoneyBadger, HB-Link stands for HoneyBadger with inter-node linking, New stands for DispersedLedger.

we run one server at each location and perform the same experiment as in § 6.2. Fig. 15 shows the results. DispersedLedger improves the throughput by at least 50% over HoneyBadger.

A.3 Example trace of temporal variation

We provide in Fig. 16 an example of the synthetic bandwidth trace we used in the temporal variation scenario in §6.3.

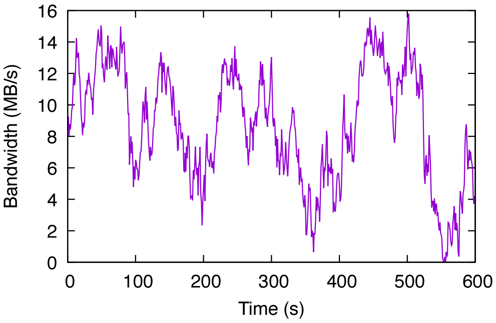


Figure 16: A bandwidth trace we used in the temporal variation scenario.

B Correctness proof of AVID-M

Notations. We use the symbol "·" as placeholders in message parameters to indicate "any". For example, Chunk$(r,\cdot,\cdot)$ means "Chunk messages with the first parameter set to r and the other two parameters set to any value".

Lemma B.1. *If a correct server sends* Ready(r)*, then at least one correct server has received* $N-f$ GotChunk(r)*.*

Proof. A correct server broadcasts Ready(r) in two cases:

1. Having received $N-f$ GotChunk(r) messages.
2. Having received $f+1$ Ready(r) messages.

If a correct server sends out Ready(r) for the aforementioned reason 1, then this already satisfies the lemma we want to prove. Now assume that a correct server sends Ready(r) because it has received $f+1$ Ready(r) (the aforementioned reason 2). Then there must exist a correct server which has sent out Ready(r) because of the aforementioned reason 1. Otherwise, there can be at most f Ready(r) messages (forged by the f Byzantine servers), and no correct server will ever send Ready(r) because of reason 2, which contradicts with our assumption. So there exists a correct server that has received $N-f$ GotChunk(r), and this satisfies the lemma. □

Theorem B.2 (Termination)**.** *If a correct client invokes* Disperse *and no other client invokes* Disperse *on the same instance of VID, then all correct servers eventually* Complete *the dispersal.*

Proof. A correct client sends correctly encoded chunks to all servers. Let's assume the Merkle root of the chunks is r, then all correct servers eventually receive Chunk$(r,\cdot,\cdot)$. Because there is no other client invoking Disperse, it is impossible for a server to receive Chunk$(r',\cdot,\cdot)$ for any $r' \neq r$, and no correct server will ever broadcast GotChunk(r') for any $r' \neq r$. So each correct server will send out GotChunk(r). Eventually, all correct servers will receive $N-f$ GotChunk(r).

All correct servers will broadcast Ready(r) upon receiving these $N-f$ GotChunk(r) messages or they have already sent Ready(r). A correct server will Complete upon receiving $2f+1$ Ready(r). We have shown that all $N-f$

correct servers will eventually send $\texttt{Ready}(r)$. Because $N-f \geq 2f+1$, all correct servers will $\texttt{Complete}$. □

Lemma B.3. *If a correct server has sent out* $\texttt{Ready}(r)$*, then no correct server will ever send out* $\texttt{Ready}(r')$ *for any* $r' \neq r$.

Proof. Let's assume for contradiction that two messages $\texttt{Ready}(r)$ and $\texttt{Ready}(r')$ $(r \neq r')$ have both been sent by correct servers. By Lemma B.1, at least one correct server has received $N-f$ $\texttt{GotChunk}(r)$, and at least one correct server has received $N-f$ $\texttt{GotChunk}(r')$ $(r' \neq r)$.

We obtain a contradiction by showing that the system cannot generate $N-f$ $\texttt{GotChunk}(r)$ messages plus $N-f$ $\texttt{GotChunk}(r')$ messages for the two correct servers to receive. Assume h $\texttt{GotChunk}(r)$ messages come from correct servers, h' $\texttt{GotChunk}(r')$ come from correct servers, and there are β Byzantine servers ($\beta \leq f$ by the definition of f). Then we have

$$h+\beta \geq N-f$$
$$h'+\beta \geq N-f.$$

A correct server do not broadcast *both* $\texttt{GotChunk}(r)$ and $\texttt{GotChunk}(r')$, while a Byzantine server is free to send different $\texttt{GotChunk}$ messages to different correct servers, so we have

$$h+h'+\beta \leq N.$$

These constraints imply

$$\beta \geq N-2f.$$

However, $\beta \leq f$, so we must have $N \leq 3f$. This contradicts with our assumption of $N \geq 3f+1$ in our security model (§2.4), so it is impossible, and the assumption must not hold. □

Theorem B.4 (Agreement)**.** *If some correct server* $\texttt{Completes}$ *the dispersal, then all correct servers will eventually* $\texttt{Complete}$ *the dispersal.*

Proof. A correct server $\texttt{Completes}$ if and only if it has received $2f+1$ $\texttt{Ready}(r)$ messages. We want to prove that in this situation, all correct servers will eventually *send* a $\texttt{Ready}(r)$, so that they will all *receive* at least $2f+1$ $\texttt{Ready}(r)$ messages needed to $\texttt{Complete}$.

We now assume a correct server has $\texttt{Completed}$ after receiving $2f+1$ $\texttt{Ready}(r)$. Out of these messages, at least $f+1$ must be broadcast from correct servers, so all correct servers will eventually receive these $\texttt{Ready}(r)$. A correct server will send out $\texttt{Ready}(r)$ upon receiving $f+1$ $\texttt{Ready}(r)$, so all correct servers will do so upon receiving the aforementioend $f+1$ $\texttt{Ready}(r)$ messages.

Because all correct servers will send $\texttt{Ready}(r)$, eventually all correct servers will receive $N-f$ $\texttt{Ready}(r)$. Because $N-f \geq 2f+1$, all of them will $\texttt{Complete}$. □

Lemma B.5. *If a correct server has* $\texttt{Completed}$*, then all correct servers eventually set the variable* $\texttt{ChunkRoot}$ *to the same value.*

Proof. A correct server uses $\texttt{ChunkRoot}$ to store the root of the chunks of the dispersed block, so we are essentially proving that all correct servers agree on this root. Assume that a server $\texttt{Completes}$, then it must have received $2f+1$ $\texttt{Ready}(r)$ messages. We now prove that no correct server can ever receive $2f+1$ $\texttt{Ready}(r')$ messages for any $r' \neq r$. Because a correct server has received $2f+1$ $\texttt{Ready}(r)$, there must be $f+1$ correct servers who have broadcast $\texttt{Ready}(r)$. By Lemma B.3, no correct server will ever broadcast $\texttt{Ready}(r')$ for any $r' \neq r$, so a correct server can receive at most f $\texttt{Ready}(r')$ for any $r' \neq r$, which are forged by the f Byzantine servers.

By Theorem B.4, all correct servers eventually $\texttt{Complete}$, so they must eventually receive $2f+1$ $\texttt{Ready}(r)$, and will each set $\texttt{ChunkRoot} = r$. □

Theorem B.6 (Availability)**.** *If a correct server has* $\texttt{Completed}$*, and a correct client invokes* $\texttt{Retrieve}$*, it eventually reconstructs some block* B'.

Proof. The $\texttt{Retrieve}$ routine returns at a correct client as long as it can collect $N-2f$ $\texttt{ReturnChunk}(r, C_i, P_i)$ messages with the same root r and valid proofs P_i. A correct server sends $\texttt{ReturnChunk}(\texttt{MyRoot},\texttt{MyChunk},\texttt{MyProof})$ to a client as long as it has $\texttt{MyRoot}$, $\texttt{MyChunk}$, $\texttt{MyProof}$, and $\texttt{ChunkRoot}$ set, and $\texttt{MyRoot} = \texttt{ChunkRoot}$. Here, a server uses $\texttt{MyRoot}$ to store the root of the chunk it has received, uses $\texttt{MyChunk}$ to store the chunk, and uses $\texttt{MyProof}$ to store the Merkle proof (Fig. 3). We now prove that if any correct server $\texttt{Completes}$, at least $N-2f$ correct servers will eventually meet this condition and send $\texttt{ReturnChunk}$ to the client.

Assume that a correct server has $\texttt{Completed}$ the VID instance with $\texttt{ChunkRoot}$ set to r. Then, by Lemmas B.4, B.5, all correct servers will eventually $\texttt{Complete}$ and set $\texttt{ChunkRoot} = r$. Also, this server must have received $2f+1$ $\texttt{Ready}(r)$ messages, out of which at least $f+1$ must come from correct servers. According to Lemma B.1, at least one correct server has received $N-f$ $\texttt{GotChunk}(r)$. At least $N-2f$ $\texttt{GotChunk}(r)$ messages must come from correct servers, so they each must have $\texttt{MyChunk}$, $\texttt{MyProof}$ set, and have set $\texttt{MyRoot} = r$.

We have proved that at least $N-2f$ correct servers will send $\texttt{ReturnChunk}(r, C_i, P_i)$ messages. For each message sent by the i-th server (which is correct), P_i must be a valid proof showing C_i is the i-th chunk under root r, because the server has validated this proof. So the client will eventually obtain the $N-2f$ chunks needed to reconstruct a block. □

Lemma B.7. *Any two correct clients finishing* $\texttt{Retrieve}$ *have their variable* $\texttt{ChunkRoot}$ *set to the same value.*

Proof. A client uses variable $\texttt{ChunkRoot}$ to store the root of the $N-2f$ chunks it uses to reconstruct the block (Fig. 4), so we are essentially proving that any two correct clients will use chunks under the same root when executing Retrieve. Let's assume for contradiction that two correct clients finish Retrieve, but have set ChunkRoot to r and r' respectively ($r \neq r'$). This implies that one client has received at least $N - 2f$ ReturnChunk$(r, \cdot, \cdot)$ messages, and the other has received $N - 2f$ ReturnChunk$(r', \cdot, \cdot)$ messages. Out of these messages, at least $N-3f$ ReturnChunk$(r,\cdot,\cdot)$ and at least $N-3f$ ReturnChunk$(r',\cdot,\cdot)$ are from correct servers (because $N \geq 3f+1$ by our security assumptions in §2.4). Since a correct server ensures MyRoot$=$ChunkRoot and uses MyRoot as the first parameter of ReturnChunk messages, there must exist some correct server with ChunkRoot set to r, and some correct server with ChunkRoot set to r'. Also, since a correct server only sends ReturnChunk when it has Completed, there must be some server which has Completed. This contradicts with Lemma B.5, which states that all correct servers must have ChunkRoot set to the same value. The assumption must not hold. □

Extra notations. To introduce the following lemma, we need to define a few extra notations. Let Encode(B) be the encoding result of a block B in the form of an array of N chunks. Let Decode(C) be the decoding result (a block) of an array of $N-2f$ chunks. Let MerkleRoot(C) be the Merkle root of an array of chunks.

Lemma B.8. *For any array of N chunks C, exactly one of the following is true:*

1. *For any two subsets D_1, D_2 of $N - 2f$ chunks in C,* Decode$(D_1)=$Decode(D_2).
2. *For any subset D of $N - 2f$ chunks in C,* MerkleRoot(Encode(Decode(D)))$\neq$MerkleRoot(C).

Proof. We are proving that a set of chunks C is either:

1. Correctly encoded (consistent), so any subset of $N-2f$ chunks in C decode into the same block.
2. Or, no matter which subset of $N-2f$ chunks in C are used for decoding, a correct client can re-encode the decoded block, compute the Merkle root over the encoding result, and find it to be *different* from the Merkle root of C, and thus detect an encoding error.

Case 1: Consistent encoding. Assume for any subset D of $N-2f$ chunks in C, Decode$(D)=B$. We now want to prove that MerkleRoot(Encode(Decode(D)))$=$MerkleRoot(C). By our assumption, Encode(Decode(D)) $=$ Encode(B), so we only need to show $C =$ Encode(B). This is clearly true by the definition of erasure code: the Encode function encodes B into a set of N chunks, of which any subset of $N-2f$ chunks will decode into B. C already satisfies this property, and the Encode process is deterministic, so it must be Encode$(B)=C$, and the lemma is satisfied in this case.

Case 2: Inconsistent encoding. Assume there exist two subsets D_1, D_2 of $N - 2f$ chunks in C, and Decode(D_1) $\neq$ Decode(D_2). Let Decode(D_1) $=$ B_1 and Decode(D_2) $=$ B_2 where $B_1 \neq B_2$. We want to prove that for any subset D of $N - 2f$ chunks in C, MerkleRoot(Encode(Decode(D)))$\neq$MerkleRoot(C).

We prove it by showing there does *not* exist any block B such that $C =$ Encode(B). That is, C is not a consistent encoding result of any block. Assume for contradiction that there exists B' such that $C =$ Encode(B'). Because D_1 is a subset of $N-2f$ chunks in C and Decode$(D_1)=B_1$, it must be $B_1 = B'$, otherwise the semantic of erasure code is broken. For the same reason $B_2 = B'$, so $B_1 = B_2$. However it contradicts with $B_1 \neq B_2$, so the assumption must not hold, and there does not exist any block B such that $C=$Encode(B).

We now prove that MerkleRoot(Encode(Decode(D)))$\neq$ MerkleRoot(C) for any subset D of $N - 2f$ chunks in C. Assume for contradiction that MerkleRoot(Encode(Decode(D))) $=$ MerkleRoot(C), then it must be that $C=$Encode(Decode(D)) because Merkle root is a secure summary of the chunks. This contradicts with the result we have just proved: there does not exist any block B such that $C=$Encode(B). So the assumption cannot hold, and the lemma is satisfied in this case. □

Theorem B.9 (Correctness)**.** *If a correct server has* Completed*, then correct clients always reconstruct the same block B' by invoking* Retrieve*. Also, if a correct client initiated the dispersal by invoking* Disperse(B) *and no other client invokes* Disperse*, then $B=B'$.*

Proof. We first prove the first half of the theorem: any two correct clients always return the same data upon finishing Retrieve. By Lemma B.7, any two clients will set their ChunkRoot to the same value. Note that a client sets ChunkRoot to the root of the chunks it uses for decoding. This implies that any two correct clients will use subsets from the *same* set of chunks. By Lemma B.8, either:

1. They both decode and obtain the same block B'.
2. Or, they each compute MerkleRoot(Encode()) on the decoded block and both get a result that is different from ChunkRoot.

In the first situation, both clients will return B'. In the second situation, they both return the block containing string "BAD_UPLOADER". In either case, they return the same block.

We then prove the second half of the theorem. Assume a correct client has initiated Disperse(B) and no other client invokes Disperse. By Theorem B.6, any correct client invoking Retrieve will obtain some block B'. We now prove that $B' = B$. Assume for contradiction that $B' \neq B$. Then the client must have received $N - 2f$ ReturnChunk(MerkleRoot(Encode(B')),$\cdot$,$\cdot$) messages. At least one of them must come from a correct server because $N-2f > f$, so at least one correct server have ChunkRoot set to MerkleRoot(Encode(B')). However, because there is only invocation of Disperse(B), all correct servers must have set ChunkRoot to MerkleRoot(Encode(B)).

```
Phase 1. Dispersal at the $i$-th server
1. For $1 \le j \le N$, let $V_i^e[j]$ be the largest epoch number $t$ such that $\mathrm{VID}_j^1, \mathrm{VID}_j^2, \ldots, \mathrm{VID}_j^t$ have Completed.
2. Let $B_i^e$ be the block to disperse (propose) for epoch $e$. $B_i^e$ contains two parts: transactions $T_i^e$ and observation $V_i^e$.
3. Invoke Disperse($B_i^e$) on $\mathrm{VID}_i^e$ as a client.
• Upon Complete of $\mathrm{VID}_j^e$ ($1 \le j \le N$), if we have not invoked Input on $\mathrm{BA}_j^e$, invoke Input(1) on $\mathrm{BA}_j^e$.
• Upon Output(1) of least $N-f$ BA instances, invoke Input(0) on all remaining BA instances on which we have not invoked Input.
• Upon Output of all BA instances,
    1. Let local variable $S_i^e \subset \{1 \ldots N\}$ be the indices of all BA instances that Output(1). That is, $j \in S$ if and only if $\mathrm{BA}_j^e$ has Output(1) at the $i$-th server.
    2. Move to retrieval phase.
Phase 2. Retrieval
1. For all $j \in S_i^e$, invoke Retrieve on $\mathrm{VID}_j^e$ to download full block $B_j^{e'}$. Decompose $B_j^{e'}$ into transactions $T_j^{e'}$ and observation $V_j^{e'}$. Let $V_j^{e'} = [\infty, \infty, \ldots, \infty]$ if $B_j^{e'}$ is ill-formatted.
2. Deliver $\{T_j^{e'} | j \in S_i^e\}$ (sorted by increasing indices). Set Delivered$[e][j] = 1$ (initially 0) for all $j \in S_i^e$.
3. For $1 \le j \le n$, let $E_i^e[j]$ be the $(f+1)^{th}$-largest value among $\{V_k^{e'}[j] | k \in S_i^e\}$.
4. For all $1 \le d \le E_i^e[j]$, check if Delivered$[d][j] = 0$. If so, invoke Retrieve on $\mathrm{VID}_j^d$ to download full block $B_j^{d'}$, and set Delivered$[d][j] = 1$ (initially 0).
5. Deliver all blocks downloaded in step 4 (sorted by increasing epoch number and node index).
```

Figure 17: Algorithm for DispersedLedger with inter-node linking. The blue color indicates the changes from the single-epoch algorithm.

So $\texttt{MerkleRoot}(\texttt{Encode}(B)) = \texttt{MerkleRoot}(\texttt{Encode}(B'))$ This contradicts with our assumption, so the assumption must not hold, and $B = B'$. □

C Specification of the full DispersedLedger protocol with Inter-node Linking

Figure 17 describes how to modify the single-epoch protocol to use inter-node linking. Blue color highlights the parts are added compared to the single-epoch protocol.

D Correctness proof of DispersedLedger

Notations. Let H ($H \subset \{1, 2, \ldots, N\}$) be the set of the indices of correct nodes. That is, $i \in H$ if and only if the i-th node is correct. In our proof, we use the variables in the full algorithm defined in Fig. 17. We also use "phase x, step y" to refer to specific steps in Fig. 17.

Lemma D.1. *For any epoch e, any $i \in H$, and any $1 \le j \le N$, if $j \in S_i^e$ then VID_i^e has* Completed *at some correct node.*

Proof. By the definition of S_i^e (phase 1, step 3), $j \in S_i^e$ if and only if BA_j^e has $\texttt{Output}(1)$ at the i-th node. By the Validity property of BA (§4.1), BA_j^e $\texttt{Output}(1)$ at a correct node implies that at least one correct node has invoked $\texttt{Input}(1)$ on BA_j^e, which only happens when that node sees VID_j^e Complete (phase 1, step 3). □

Lemma D.2. *For any epoch e, any $i, j \in H$, $S_i^e = S_j^e$ and $E_i^e = E_j^e$.*

Proof. By the definition of S_i^e (phase 1, step 3), $k \in S_i^e$ if and only if BA_k^e has $\texttt{Output}(1)$ at the i-th node. By the Agreement property of BA (§4.1), BA_k^e will eventually $\texttt{Output}(1)$ at the j-th node. So $k \in S_i^e$ if and only if $k \in S_j^e$, and $S_i^e = S_j^e$.

We now prove $E_i^e = E_j^e$. The i-th node (which is correct) starts the computation of E_i^e by invoking Retrieve on all VIDs in $\{\mathrm{VID}_k^e | k \in S_i^e\}$. These Retrieves are guaranteed to finish by Lemma D.1 and the Availability property of VID (Theorem B.6). The node then extracts the observations $\{V_k^{e'} | k \in S_i^e\}$ from the downloaded blocks. Note that the j-th node will download the same set of observations. This is because $S_i^e = S_j^e$, and the VID Correctness property (Theorem B.9) guarantees the j-th node will obtain the same blocks when invoking Retrieve on $\{\mathrm{VID}_k^e | k \in S_j^e\}$.

To combine the observations into the estimation, the i-th node runs phase 2, step 3. This process is deterministic, with E_i^e being a function of the observations $\{V_k^{e'} | k \in S_i^e\}$ and parameter f. Because we have just proved the j-th node will obtain the same set of estimations, and by our security model f is a protocol parameter known to all nodes (§2.4), the j-th node will get the same results.

□

Lemma D.3. *For any epoch e, and any $i \in H$, $|S_i^e| \ge N - f$. That is, S_i^e contains at least $N - f$ indices.*

Proof. By the definition of S_i^e (phase 1, step 3), this lemma essentially states that at least $N - f$ BAs among $\{\mathrm{BA}_1^e, \mathrm{BA}_2^e, \ldots, \mathrm{BA}_N^e\}$ will $\texttt{Output}(1)$ at the i-th node.

Assume for contradiction that $|S_i^e| < N - f$. By Lemma D.2, $|S_j^e| < N - f$ for all $j \in H$, i.e., less than $N - f$ BAs eventually $\texttt{Output}(1)$ at every correct node. We now consider the possible outcomes of the remaining BA instances, which do not eventually $\texttt{Output}(1)$.

One possibility is some of them $\texttt{Output}(0)$. According to phase 1, step 3, correct nodes will not invoke $\texttt{Input}(0)$ on any BA instance unless $N - f$ BA instances have $\texttt{Output}(1)$. By our assumption, less than $N - f$ BA $\texttt{Output}(1)$, so the latter is not happening and *no* correct nodes will $\texttt{Input}(0)$ on *any* BA instance. By the Validity property of BA (§4.1), no BA instance can $\texttt{Output}(0)$.

We have showed that the remaining BAs cannot $\texttt{Output}(0)$, so it must be that all of them never terminate. We will prove it is also impossible. Assume for contradication that all BAs that do not $\texttt{Output}(1)$ never terminate. By our assumption, less than $N-f$ BAs `Output(1)`, so there must exist $k \in H$ such that BA$_k^e$ never terminates. By the Termination property of VID (Theorem B.2), VID$_k^e$ eventually `Completes` on all correct nodes. According to phase 1, step 3, because not all BAs will terminate, all correct nodes will stay at this step. All correct nodes will `Input(1)` to BA$_k^e$ upon seeing VID$_k^e$ `Complete`. By the Termination and Validity properties of BA (§4.1), BA$_k^e$ will terminate and `Output(1)`, which conflicts with our assumption.

We have showed there is no valid outcome for the remaining BA instances, so our assumption cannot hold, and at least $N-f$ BA instances eventually `Output(1)` at all correct nodes. □

Lemma D.4. *For any epoch e, any $i \in H$, and any $1 \le j \le N$, there exist $p,q \in H$ such that $V_p^e[j] \le E_i^e[j] \le V_q^e[j]$.*

Proof. The lemma states that if the i-th node (which is correct) computes the estimation $E_i^e[j]$ for the j-th node, then the estimation is lower- and upper-bounded by the *observations* $V_p^e[j]$ and $V_q^e[j]$ of two correct nodes (with indices p and q). That is, the estimation is not too high or too low.

Now assume for contradication that for some $1 \le j \le N$, for all $p \in H$, $V_p^e[j] > E_i^e[j]$. That is, the estimation for j is not lower bounded by the observations made by any correct node. According to phase 2, step 3, the i-th node sets $E_i^e[j]$ to the $(f+1)^{th}$-largest value among $\{V_k^{e\prime}[j] | k \in S_i^e\}$. Here, $V_k^{e\prime}$ is the observation of the k-th node downloaded by invoking `Retrieve` on VID$_k^e$. By Lemma D.1 and VID Availability property (Theorem B.6), the `Retrieves` will eventually finish.

By our assumption, for all $p \in H \cap S_i^e$, $V_p^e[j] > E_i^e[j]$. By the VID Correctness property (Theorem B.9), the observations of correct nodes will be correctly downloaded. That is, $V_k^{e\prime} = V_k^e$ for all $k \in H$. So for all $p \in H \cap S_i^e$, $V_p^{e\prime}[j] > E_i^e[j]$. By Lemma D.3, $|S_i^e| \ge N-f$, so $|H \cap S_i^e| \ge N-2f$. So there are *at least* $N-2f$ values in $\{V_k^{e\prime}[j] | k \in S_i^e\}$ that are greater than $E_i^e[j]$. However, $E_i^e[j]$ is the $(f+1)^{th}$-largest value among $\{V_k^{e\prime}[j] | k \in S_i^e\}$, so there can be *at most* f values in $\{V_k^{e\prime}[j] | k \in S_i^e\}$ that are greater than $E_i^e[j]$. Because $N > 3f$ (§2.4), $N-2f > f$, so the two conclusions are in conflict, and the assumption cannot hold.

We can similarly prove it is impossible that for some $1 \le j \le N$, for all $q \in H$, $V_q^e[j] < E_i^e[j]$. □

Theorem D.5 (DispersedLedger is well-defined)**.** *For any epoch e, any $i \in H$, the i-th node eventually finishes epoch e.*

Proof. This lemma states that correct nodes will never be stuck in any epoch e, so that our algorithm is well-defined. To prove that, we go through Fig. 17 line by line and prove each step will eventually finish.

Phase 1, steps 1–2. These are local computation and will finish instantly.

Phase 1, step 3. This step finishes as soon as all BA instances in that epoch `Output`. By Lemma D.3, all correct nodes eventually see at least $N-f$ BA instances `Output(1)`. At that point, each correct node will invoke `Input(0)` into all BAs on which it has not invoked `Input`. This ensures that all correct nodes eventually invoke `Input` on all BAs. By the Termination property of BA (§4.1), all BAs will eventually `Output` on all correct nodes, which ensures this step will finish.

Phase 2, step 1. This step finishes as soon as `Retrieves` on $\{\text{VID}_j^e | j \in S_i^e\}$ finish. By Lemma D.1, $\{\text{VID}_j^e | j \in S_i^e\}$ will `Complete` on all correct nodes. Then by VID Availability property (Theorem B.6), the `Retrieves` will finish.

Phase 2, steps 2–3. These are local computation and will finish instantly.

Phase 2, step 4. This step will finish if for all $1 \le j \le N$, for all $1 \le d \le E_i^e[j]$, `Retrieve` of VID$_j^d$ finishes. By Lemma D.4, there exists $q \in H$ such that $V_q^e[j] \ge E_i^e[j]$, and the q-th node (which is correct) reports that VID$_j^t$ has `Completed` for all $1 \le t \le V_q^e[j]$. By VID Availability property (Theorem B.6), the `Retrieves` will eventually finish, so this step will finish.

Phase 2, steps 5. This is local computation and will finish instantly. □

Theorem D.6 (Validity)**.** *All blocks proposed by correct nodes are eventually delivered by all correct nodes.*

Proof. Assume the i-th node (which is correct) proposes block B_i^e in epoch e. The i-th node invokes `Disperse`(B_i^e) on VID$_i^e$. By VID Termination property (Theorem B.2), eventually all correct nodes will see VID$_i^e$ `Complete`. So there must exist an epoch t where for all $j \in H$, $V_j^t[i] \ge e$. That is, in epoch t, all correct nodes report that the i-th node has at least dispersed into VID$_i^1$ to VID$_i^e$. By Lemma D.4, for all $j \in H$, $E_j^t[i] \ge e$. According to phase 2, steps 4–5, all correct nodes either have already delivered B_i^e in previous epochs, or will deliver B_i^e in epoch t. □

Theorem D.7 (Agreement and Total Order)**.** *Two correct nodes deliver the same sequence of blocks.*

Proof. Let $i,j \in H$. We prove this theorem by induction on the number of epochs the i-th and the j-th nodes have finished. In other words, we prove that for any $t \ge 0$, the i-th and the j-th nodes deliver the same sequence of blocks in the first t epochs.

Initial ($t=0$). Both nodes have not delivered any block. So the hypothesis clearly holds in this situation.

Induction step. Assume our hypothesis holds for $t = e-1$ ($e \ge 1$). We now prove the hypothesis holds for $t = e$. We first show the two nodes commit the same sequence of blocks with BA. By Lemma D.2, $S_i^e = S_j^e$ and $E_i^e = E_j^e$. According to phase 2, step 1, both nodes will invoke `Retrieve` on the same set of VIDs. By VID Correctness property (Theorem B.9), they will get the same set of blocks and deliver them in the same order in phase 2, step 2.

We now show the two nodes commit the same sequence of blocks with inter-node linking. The local variable `Delivered` stores whether a node has delivered a block (phase 2, steps 2, 4). By the induction hypothesis, the two nodes have delivered the same sequence of blocks prior to epoch e, so the variable `Delivered` is the same on the two nodes. By Lemma D.2, $E_i^e = E_j^e$. So the two nodes will invoke `Retrieve` on the same set of VIDs in phase 2, step 4 and get the same set of blocks. Both nodes sort the blocks deterministically and deliver them in the same order in phase 2, step 5.

We have proved that the i-th and the j-th nodes deliver the same sequence of blocks in epoch e. By our induction hypothesis, they deliver the same sequence until epoch $e-1$. So they deliver the same sequence in the first e epochs. This completes the induction. □