The Distributed Producer-Consumer Model Is All You Need: A Compiler for Asynchronous BFT

Liang Zhang*, Binyang Hou†, Zhengxuan Guo*, Xiaohai Dai‡, Haibin Kan†, Jiheng Zhang*, Moti Yung§
*The Hong Kong University of Science and Technology
†Fudan University
‡Huazhong University of Science and Technology
§Google / Columbia University
briliasm@gmail.com, 25113050255@m.fudan.edu.cn, zguocd@connect.ust.hk
xhdai@hust.edu.cn, hbkan@fudan.edu.cn, jiheng@ust.hk, motiyung@gmail.com

Abstract*—Building an Asynchronous Byzantine Fault Tolerant (BFT) protocol requires establishing payload availability with total order despite Byzantine faults and arbitrarily delayed messages. Although many BFT systems separate payload dissemination from agreement, they often couple their composition to protocol-specific selection or scheduling mechanisms. We present the Distributed Producer-Consumer Model (DPCM), an event-driven abstraction that separates payload-specific certified availability from payload-oblivious binary finalization. We prove that DPCM compiles any producer–consumer pair satisfying interfaces specified herein into an asynchronous BFT object log. DPCM batch-finalizes certified heights, deterministically refining failed batches until each height yields a normal or evidence-backed dummy entry. We instantiate DPCM as Octopus, which combines parallel consistent broadcast producers with batched asynchronous binary agreement (ABA). To achieve performance advantages, Octopus lets producers certify blocks without waiting for consumer decisions and keeps payload recovery off the agreement path. One ABA optimistically finalizes multiple consecutive producer columns, while failures trigger parallel row-wise refinement with logarithmic bisection depth. Across an eight-region WAN with $n \in \{8, 16, 32\}$ and 2200-transaction blocks, Octopus remains within 6.5% of Dumbo-NG's (ACM CCS'22) fault-free throughput. With the maximum tested number of silent faulty parties, it outperforms Dumbo-NG by 25.9% and 18.7% at $n = 8$ and $n = 16$, respectively, and remains within 1% at $n = 32$. At $n = 16$ and $n = 32$, Dumbo-NG's latency is 1.9–2.5× that of Octopus. Across all evaluated settings, Octopus achieves 2.9–16.8× the throughput of Alea-BFT (USENIX NSDI'24), whose latency is 2.4–15.2× that of Octopus.

I. Introduction

Asynchronous Byzantine fault tolerance (BFT) maintains a common continuously growing log despite Byzantine faults and arbitrarily delayed messages [26], [32]. The classical FLP impossibility [17] makes randomization necessary for asynchronous consensus progress. BFT cost, in turn, comes from both disseminating large payloads and agreeing on a common order, while ensuring that every finalized payload remains retrievable. Modern protocols therefore separate payload dissemination from agreement in order to improve modularity and performance [4], [18], [28]. Dissemination establishes that an accepted object is uniquely determined and retrievable, whereas agreement determines which log heights become final. This separation keeps large payloads off the agreement path. The main challenge is, in fact, ensuring a correct handoff from availability to finalization.

Many asynchronous BFT constructions use asynchronous common subset (ACS) [5], [16], [20], [26] as a per-epoch agreement abstraction. Each of the n parties proposes a value, and all correct parties output the same subset containing proposals from at least $n - f$ parties, where f is the number of Byzantine faults the system can tolerate. Deterministically ordering and appending successive ACS outputs produces a common log. Thus, implementing ACS suffices to realize asynchronous BFT protocols. The canonical BKR-style ACS construction runs n reliable-broadcast (RBC) instances [8] to disseminate proposals and n asynchronous binary agreement (ABA) instances to decide their inclusion [5], [26]. Cachin et al. formalized the multi-valued Byzantine agreement with external validity [33]. In multi-valued validated Byzantine agreement (MVBA), parties propose values from a non-binary domain and agree on the same proposed value satisfying a public validity predicate [1], [24]. MVBA yields an alternative ACS reduction: after n RBC instances guarantee publicly verifiable completion evidence, parties can propose certificate vectors covering at least $n - f$ RBC outputs to one MVBA instance [20], [21], [24]. Despite their different agreement approaches, both ACS constructions depend on the same fundamental handoff: dissemination must provide sufficient evidence for agreement to safely finalize available objects.

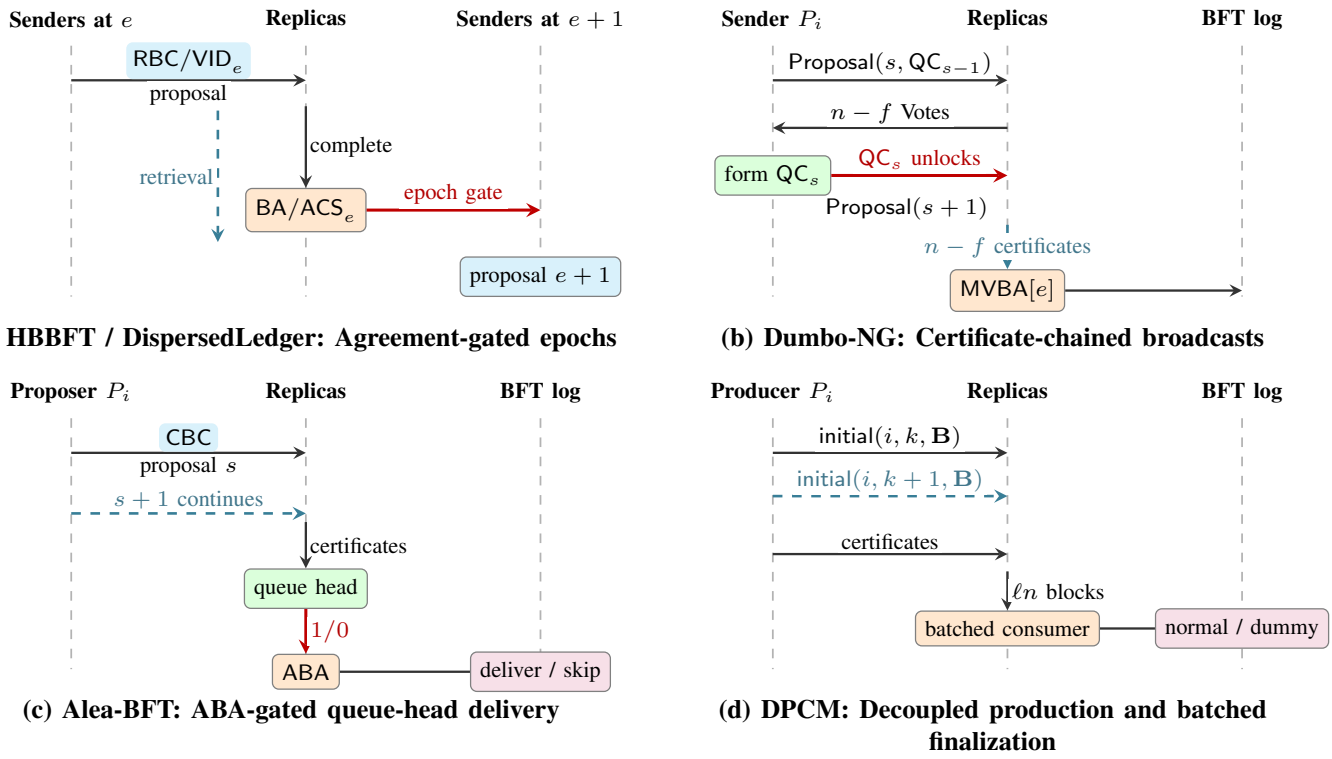
Representative asynchronous BFT designs gate epochs on agreement [26], [28], chain broadcasts through quorum certificates [18], or use ABA to gate queue-head delivery [4]. Figure 1 illustrates their dissemination–agreement dependencies. This paper asks:

what interfaces suffice to compose payload-oblivious objects and agreement into a BFT log that eventually resolves every position, without requiring certificate chains or forcing production to wait for agreement?

We further ask:

whether the interface can safely finalize multiple log positions with only a single binary agreement instance?

We answer the first question with the Distributed Producer-Consumer Model (DPCM), an event-driven composition abstraction (akin of middleware) for asynchronous BFT object



(a) HBBFT / DispersedLedger: Agreement-gated epochs

(b) Dumbo-NG: Certificate-chained broadcasts

(c) Alea-BFT: ABA-gated queue-head delivery

(d) DPCM: Decoupled production and batched finalization

Fig. 1. Simplified message- and subprotocol-level dependencies in HBBFT/ DispersedLedger [26], [28], Dumbo-NG [18], Alea-BFT [4], and DPCM. Panel (a) groups HBBFT and DispersedLedger only by their agreement-gated epoch progression. Red arrows denote dependencies that gate subsequent dissemination or determine an agreement input.

logs. Its producer protocol Π_P assigns objects to producer slots and issues publicly verifiable certificates that bind each certified slot to a recoverable object, while its payload-oblivious consumer protocol Π_C finalizes log heights using bits and verifiable decision evidence. A public placement algorithm σ maps each producer slot to one log height. A decision 1 yields a certified normal entry, whereas a decision 0 yields an evidence-backed dummy entry. Our composition theorem shows that producer-side verifiability, binding, and recoverability, together with consumer-side agreement, binary validity, and eventual coverage of every height, suffice to obtain a BFT object log. The DPCM compiler neither schedules nor serializes the internal execution of the two protocols, and its proof is independent of the broadcast transcript, erasure code, and payload semantics.

We answer the second question with a semantics-preserving batched consumer. A correct party inputs 1 to a batch of slots when it holds certificates for every slot in the batch. A decision 1 therefore resolves every covered height as normal. A decision 0 for a non-singleton batch resolves nothing. Instead, its decision evidence authorizes deterministic concurrent refinement. Only a singleton decision 0 yields a dummy entry. This rule preserves the certificate, decision-evidence, recovery, and ordering semantics of every height while allowing one binary agreement instance to resolve many certified slots in the optimistic path.

We instantiate both results as Octopus, an asynchronous BFT state-machine replication (SMR) protocol. The producers continuously leverage parallel consistent-broadcast (CBC) [3], [20] to certify predetermined queue positions without an agreement-epoch barrier, while the consumer jointly finalizes their assigned log heights in batches using ABA protocols [7], [46]. A root batch covers ℓ columns of n positions, with each column comparable in output size to one ACS epoch. In the optimistic path, only a single ABA instance finalizes all ℓn positions. If the root batch decides 0, Octopus partitions it into n row batches, one for each producer, and invokes these row-level agreement instances concurrently. This parallelism allows a row whose certificates have reached all correct parties to decide 1 and finalize all its ℓ positions at once, without waiting for rows affected by delayed messages or faulty producers. A row segment that decides 0 is bisected into two child batches, which are again processed in parallel. Refinement continues only along segments that decide 0, thereby isolating delayed or unavailable positions while retaining batched finalization for ready segments. In the worst case, a faulty producer may leave every position in its row uncertified, causing every non-singleton segment in that row to decide 0 and the refinement to reach all ℓ singleton slots. The resulting full binary refinement tree contains at most $2\ell - 1$ agreement instances for that row, which is the maximum agreement work induced by a single producer row. Section V-A analyzes the costs of Octopus's optimistic and pessimistic (worst-case) paths, and Section V-B evaluates its performance in a wide-area network

(WAN) deployment. Table I compares Octopus's asymptotic costs per n-block (ACS) output with related asynchronous BFT protocols.

TABLE I
EXPECTED ASYMPTOTIC COST PER n-BLOCK OUTPUT, ACCOUNTING FOR RANDOMIZED RECOVERY

Protocol	Latency	Communication	Messages
HBBFT [26]	$O(\log n)$	$O(n^2(\lvert\mathbf{B}\rvert + \lambda n \log n))$	$O(n^3)$
Dumbo [20]	$O(1)$	$O(n^2(\lvert\mathbf{B}\rvert + \lambda n \log n))$	$O(n^3)$
Dumbo-NG [18]	$O(1)$	$O(n^2(\lvert\mathbf{B}\rvert + \lambda n \log n))$	$O(n^2)$
ALEA-BFT [4]	$O(n)$	$O(n^3(\lvert\mathbf{B}\rvert + \lambda))$	$O(n^3)$
Octopus (worst case)†	$O(\log n + \log(\ell+1))$	$O(n^2(\lvert\mathbf{B}\rvert + \lambda n))$	$O(n^3)$
Octopus (optimistic)	$O(1)$	$O(n^2(\lvert\mathbf{B}\rvert + \lambda))$	$O(n^2)$

†: The bound assumes $c = f = \Theta(n)$, where c is the number of fully refined producer rows, and uses the expected cost of randomized recovery.

This paper presents the following contributions:

- *A composition abstraction and theorem.* We introduce DPCM, which specifies the guarantees required at the boundary between certified object availability and binary height finalization. We prove that any producer and consumer satisfying these interfaces compose into an asynchronous BFT object log in which every predetermined height eventually becomes either a recoverable normal entry or an evidence-backed dummy.
- *Batch consumer with deterministic refinement.* We extend the consumer so that one binary agreement decision can finalize multiple predetermined heights when all their producer certificates are available. A decision 0 on a non-singleton batch finalizes no dummy entries; it instead authorizes deterministic parallel refinement until every height is resolved. We prove that batching preserves DPCM's per-height agreement, validity, availability, and ordering guarantees.
- *The* Octopus *asynchronous SMR protocol.* We instantiate DPCM, as Octopus, with continuously running CBC producers and a batched ABA consumer. Producers do not wait for agreement before proposing subsequent blocks, while look-ahead invocation and row-wise refinement prevent missing producer certificates from blocking log progress. The optimistic path finalizes ℓn positions with one ABA instance, giving expected $O(n^2)$ messages and $O(n^2(|\mathbf{B}| + \lambda))$ bits per column with expected $O(1)$ consumer latency. We prove safety and probability-one progress without timeouts.
- *An extensive WAN evaluation.* We implement Octopus, Dumbo-NG [18], and Alea-BFT [4] in a common Go framework that shares networking, cryptographic, workload, and measurement components. The evaluation covers eight Alibaba Cloud regions, $n \in \{8, 16, 32\}$ parties, six block sizes, and both fault-free executions and executions with the maximum tested number of silent faulty parties. A 72-setting sensitivity study characterizes how the consumer batch size and producer-queue length govern the throughput–latency trade-off. Across the tested scales and fault regimes, Octopus sustains Dumbo-NG-level throughput, provides lower commit latency at $n = 16$ and $n = 32$, and consistently outperforms Alea-BFT on both metrics.

II. RELATED WORK

Work closest to DPCM deals with how dissemination establishes object availability and how agreement determines which available objects become final.

Asynchronous Byzantine agreement. Pease et al. formalized agreement in the presence of faults [48], and Schneider described how replicas executing the same requests in the same order implement SMR [50]. For asynchronous networks, Ben-Or and Rabin introduced randomized agreement protocols [31], [49], and Canetti and Rabin later achieved optimal resilience in an expected constant number of asynchronous rounds [34]. Bracha and Toueg [32] connected asynchronous consensus and broadcast, and subsequent work developed practical cryptographic, optimal-resilience, signature-free, and adaptively secure ABA constructions [11], [30], [46], [47]. Cobalt [25] adapts Mostéfaoui et al.'s ABA [46] to non-uniform trust, adding confirmation for its weaker common random source and a finish phase that makes consistency local, while ensuring probabilistic progress under its connectivity assumptions.

ACS- and MVBA-based asynchronous BFT. HBBFT implements asynchronous atomic broadcast by repeatedly invoking ACS, with n reliable-broadcast instances disseminating proposals and n ABA outputs determining their inclusion [26]. Dumbo2 instead combines n provable broadcasts with one MVBA over a vector containing valid completion proofs for at least $n - f$ broadcasts [20]. Subsequent constructions improve MVBA or ACS communication, parallelize ABA instances, or avoid signatures and public-key infrastructure under their respective assumptions [21], [24], [36], [37], [55]. These constructions differ in their completion evidence, validity predicates, and recovery requirements.

Pipelined dissemination and ordering. Several protocols move payload dissemination outside the main agreement path. Dumbo-NG [18] continuously runs one broadcast chain per sender: the proposal for slot $s+1$ carries the quorum certificate for slot s, and MVBA periodically agrees on vectors of certified chain positions. JUMBO [13] retains this structure but disperses and aggregates MVBA certificate vectors to reduce authenticator overhead at large scales. Alea-BFT [4] runs CBC concurrently with agreement, stores delivered proposals in per-replica priority queues, and uses ABA to decide whether the head selected from the current queue is delivered. HBBFT [26] instead executes consecutive ACS epochs. DispersedLedger [28] runs n verifiable information dispersal (VID) and n binary agreement (BA) instances in each epoch, starts the next epoch after the current epoch's BA instances finish, and allows payload retrieval to overlap later epochs.

Optimistic asynchronous atomic broadcast combines a fast normal path with asynchronous fallback [43]. Bolt-Dumbo

Transformer [44] combines a deterministic fastlane with asynchronous fallback, while Icarus [15] rotates broadcast chains through the optimistic role. Stratus [40] decouples transaction dissemination from leader-driven agreement through a provably available shared mempool. ISS [53] instead scales leader-driven SMR by assigning disjoint log segments to concurrent sequenced-broadcast instances.

Directed-acyclic-graph (DAG)-based protocols [14], [39], [41], [42], [51] instead derive atomic broadcast by ordering disseminated vertices through their causal structure.

Certified availability, dispersal, and recovery. Reliable broadcast (RBC) provides validity, agreement, and totality [3], [8], [52]. Consistent broadcast (CBC) retains validity and agreement but omits totality [3], [20]. Provable broadcast additionally returns succinct completion evidence. Das et al.'s asynchronous data dissemination [35] starts from data held by sufficiently many correct parties and disseminates it to all correct parties. Speeding-Dumbo [21] uses provable broadcast with a separate recovery phase and applies information dispersal to bound the communication of worst-case recovery. These differences make it important to state precisely what a producer certificate proves and what recovery is still required.

Information dispersal [27] encodes a payload into fragments from which sufficiently many fragments reconstruct the original value. Asynchronous verifiable information dispersal (AVID) [9] adds consistency and availability properties in an asynchronous Byzantine setting. BEAT3 [16] replaces the reliable-broadcast component of an ACS construction with fingerprinted asynchronous verifiable information dispersal (AVID-FP) [22], which uses fingerprinted cross-checksums, so that ACS orders compact metadata rather than full payloads. Yang et al.'s DispersedLedger introduces AVID-M, whose all-to-all control messages carry a constant-size Merkle root and whose retrieval procedure detects inconsistent encoding by re-encoding the decoded value [28]. DispersedLedger combines one AVID-M and one BA instance per proposer in each epoch and retrieves the selected payloads after agreement. These are concrete mechanisms for ordering erasure-coded data without placing full-payload retrieval on the agreement path.

BFT storage. Prior BFT storage systems [6], [10], [19], [45] implement read/write register abstractions using Byzantine quorums, versioning, replication, or erasure coding. MVP-ORAM [54] adds wait-free concurrent access-pattern privacy over confidential BFT storage, a service concern orthogonal to DPCM's append-only log. BEAT3 [16] is closest to our append-only instantiation: it uses AVID-FP [22] to store coded fragments and ACS to order their fingerprinted cross-checksums. Appendix D instead uses DPCM's predetermined positions and normal-or-dummy finalization.

Positioning of DPCM. Prior systems already separate dissemination from agreement, but couple their composition to subset selection, certificate vectors, broadcast chains, or protocol-specific queues. DPCM instead formalizes the availability and binary-finalization guarantees sufficient for composition, assigns objects to predetermined heights, and resolves every height as normal or dummy. This interface allows one decision to cover several heights without depending on a particular broadcast or dispersal mechanism.

III. The DPCM Compiler

A. Overview

The goal of distributed producer-consumer model (DPCM) is to compose certified object availability with payload-oblivious binary finalization into an asynchronous BFT object log. A producer process Π_P certificate binds a recoverable object to a producer slot, while a consumer Π_C decision finalizes that height as normal or dummy. The DPCM compiler connects the two processes while allowing them to execute independently.

DPCM connects the producer and consumer namespaces through the public deterministic placement algorithm σ. Each object originates at a producer slot $s = (i, k)$, denoting producer P_i's k-th position, while the consumer addresses the output log by height h. The placement algorithm assigns s to the unique height $h = \sigma(s)$. A producer certificate π binds s to an object commitment c. At height h, consumer decision $d = 1$ yields a normal entry only when the certificate for $\sigma^{-1}(h)$ verifies and the committed object is recoverable. Decision $d = 0$ yields a dummy no-op. Consequently, absent producer material cannot indefinitely block log progress.

The DPCM abstraction is defined by interfaces rather than tied to particular dissemination or agreement primitives. The producer must satisfy *certified availability*, and the consumer must satisfy *binary height finalization*. The DPCM compiler, denoted Compile, connects the public events of Π_P and Π_C and emits ordered log entries without controlling their internal execution. Any pair of protocols satisfying these interfaces can therefore be composed as

$$\mathsf{Compile}(\Pi_P, \Pi_C) \Longrightarrow \textbf{BFT Object Log}.$$

B. System Model and Core Terms

1) System model: We consider $n = 3f + 1$ parties in an authenticated asynchronous message-passing network, with at most f Byzantine parties. Unlike the partial-synchrony model of Dwork et al. [38], there is no bound on message delay, but messages between correct parties are eventually delivered. Cryptographic failures occur only with negligible probability, and randomized agreement terminates with probability one. All protocol messages, certificates, and agreement instances are scoped to the same public session and membership.

2) Producer slots and consumer heights: Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the party set and let $\mathbb{N}^+$ denote the positive integers. A *producer slot* $s = (i, k)$ is the k-th position of producer P_i. The producer slot domain is $\mathcal{S} = \mathcal{P} \times \mathbb{N}^+$. A *height* $h \in \mathbb{N}^+$ is a position in the consumer's finalized object log. Accordingly, Π_P addresses objects by slots, whereas Π_C finalizes the resulting log by height. The consumer runs the public placement algorithm $\sigma : \mathcal{S} \to \mathbb{N}^+$. We require σ to be a bijection: $\sigma(s)$ is the unique height assigned to slot s, and $\sigma^{-1}(h)$ is the unique producer slot assigned to height h.

3) Objects and certificates: An object O is represented by a commitment $c = \mathsf{Commit}(O)$, such as a collision-resistant digest of a block or authenticated storage metadata. We require Commit to be computationally binding. A producer certificate π is a publicly verifiable evidence that Π_P has certified commitment c for producer slot s. It may be instantiated by a quorum certificate, broadcast proof, or data-availability certificate.

C. Producer and Consumer Interfaces

DPCM exposes Π_P and Π_C through event-driven interfaces. The two protocols execute independently and do not invoke each other. Each emits events handled by the compiler without serializing their execution or introducing a shared scheduler. These events describe externally observable protocol transitions rather than internal implementation steps. This design gives the DPCM event interface two complementary roles. Formally, it states the properties sufficient for the composition theorem. Operationally, it lets an implementation replace the producer's dissemination and recovery mechanism or the consumer's agreement mechanism while preserving the same events and properties, without changing the compiler or the object-log interface.

Definition 1 (Producer: Certified Availability). For each party P_i, the local producer protocol Π_P exposes three events:

$$\textsc{Produce}(s, O), \quad \textsc{Certified}(s, c, \pi),$$
$$\textsc{Recover}(s, c, \pi) \rightsquigarrow O.$$

The event $\textsc{Produce}(s, O)$ supplies object O to Π_P for certification at producer slot s. Each party produces only into its own slots. At P_i, the slot must have the form $s = (i, k)$. Protocol Π_P emits Certified when it locally creates or accepts a certificate π binding any producer slot s to commitment c. The event Recover requests the certified object O. Protocol Π_P satisfies:

- *Public verification*: Any party can evaluate the predicate $\mathbf{ValidProd}(s, c, \pi)$, which returns 1 exactly when π is an accepted producer certificate for (s, c).
- *Slot binding*: Valid certificates for the same slot s cannot bind two different commitments.
- *Certified availability*: Any correct party invoking its local Recover on a valid (s, c, π) eventually obtains an O with $\mathsf{Commit}(O) = c$.

Definition 2 (Consumer: Binary Height Finalization). The local consumer protocol Π_C exposes three events:

$$\textsc{Ready}(h), \quad \textsc{Consume}(h, b), \quad \textsc{Decided}(h, d, e),$$

Here h is the log height, $b \in \{0, 1\}$ is a party's input bit, $d \in \{0, 1\}$ is the finalized bit, and e is decision evidence. At a correct party, $\textsc{Ready}(h)$ requests one local input for height h. A party responds once with $\textsc{Consume}(h, b)$, where $b = 1$ iff a valid producer certificate π for $s = \sigma^{-1}(h)$ is locally available. These inputs drive asynchronous binary finalization for height h. Upon termination, Π_C emits $\textsc{Decided}(h, d, e)$. A decision $d = 1$ denotes a normal height, whereas $d = 0$ denotes a dummy height. The value e is publicly verifiable decision evidence. Protocol Π_C satisfies:

- *Agreement*: Each correct party emits at most one irrevocable decision for height h, and no two correct parties emit different decisions.
- *Coverage*: For each height h, every correct party eventually emits exactly one $\textsc{Ready}(h)$, no matter whether the corresponding producer slot becomes certified.
- *Termination*: Once all correct parties submit $\textsc{Consume}(h, b)$, they emit $\textsc{Decided}(h, d, e)$ with probability one.
- *Binary validity*: If all correct parties input the same bit $b \in \{0, 1\}$, then the decision is $d = b$. Equivalently, every decided bit was input by at least one correct party.
- *Decision evidence*: Any party can evaluate the predicate $\mathbf{ValidDec}(h, d, e)$. Every decision emitted by a correct party carries evidence accepted by this predicate. Evidence accepted for both $d = 0$ and $d = 1$ cannot exist for the same height h.
- *Ordered decision*: Consumer decisions may be computed out of height order, but at a correct party, Π_C emits $\textsc{Decided}(h, d, e)$ only after the local log has delivered through height $h - 1$.

D. BFT Object Log and DPCM Compiler

The two interfaces provide complementary evidence: Π_P binds a producer slot s to a recoverable object through (c, π), while Π_C finalizes a log height h through (d, e). For each height, the DPCM compiler uses $s = \sigma^{-1}(h)$ to combine these outputs and emits $\textsc{Deliver}(h, \mathsf{entry})$ at each party. The resulting outputs form the BFT object log formalized below.

Definition 3 (BFT Object Log). Let $\mathcal{L}_i$ be the local log formed by the entries delivered at P_i. A normal entry is (O, π, e), while a dummy entry $(\perp, e)$ is a deterministic no-op. A BFT object log is the collection of correct-party logs $\{\mathcal{L}_i\}$ satisfying:

- *Integrity*: A correct party delivers at most one immutable entry at each height.
- *Agreement*: No two correct parties deliver semantically different entries at the same height.
- *Total order*: A correct party delivers entries in increasing height order without gaps.
- *External validity*: For $s = \sigma^{-1}(h)$, a normal entry (O, π, e) satisfies $\mathbf{ValidProd}(s, \mathsf{Commit}(O), \pi) = 1$ and $\mathbf{ValidDec}(h, 1, e) = 1$. A dummy entry $(\perp, e)$ satisfies $\mathbf{ValidDec}(h, 0, e) = 1$.
- *Data availability*: If a correct party delivers a normal entry, every correct party that requests its object eventually obtains the same O.
- *Termination*: For every fixed height h, every correct party eventually delivers a normal or dummy entry at h.

Deliver is the output boundary of DPCM. Algorithm 1 implements the DPCM compiler at P_i. Lines 1–2 initialize the shared certificate map and start Π_P and Π_C concurrently. Lines 3–6 accept only valid certificates and broadcast each

certificate before installing it. Thus, any certificate supporting an input 1 is already being disseminated. On $\textsc{Ready}(h)$, Lines 7–10 map h to its producer slot and input whether its certificate is locally available. Finally, Lines 11–22 verify the decision evidence and deliver either a dummy or a recovered and verified normal object.

Algorithm 1 $\mathsf{Compile}(\Pi_P, \Pi_C)$ at party P_i

1: $cert \leftarrow \emptyset$
2: start Π_P and Π_C concurrently
3: **upon** Π_P emits $\textsc{Certified}(s, c, \pi)$ or receives (s, c, π)
4: **if** $cert[s]$ is undefined and $\mathbf{ValidProd}(s, c, \pi) = 1$ **then**
5: broadcast (s, c, π)
6: $cert[s] \leftarrow (c, \pi)$
7: **upon** Π_C emits $\textsc{Ready}(h)$
8: $s \leftarrow \sigma^{-1}(h)$
9: $b \leftarrow 1$ iff $cert[s]$ is defined; otherwise $b \leftarrow 0$
10: submit $\textsc{Consume}(h, b)$ to Π_C
11: **upon** Π_C emits $\textsc{Decided}(h, d, e)$
12: verify $\mathbf{ValidDec}(h, d, e) = 1$
13: **if** $d = 0$ **then**
14: $\textsc{Deliver}(h, (\perp, e))$
15: **else**
16: invoke $\mathsf{CompleteNormal}(h, e)$ asynchronously
17: **func** $\mathsf{CompleteNormal}(h, e)$
18: $s \leftarrow \sigma^{-1}(h)$
19: wait until $cert[s] = (c, \pi)$ is defined
20: $O \leftarrow \textsc{Recover}(s, c, \pi)$
21: verify $\mathsf{Commit}(O) = c$
22: $\textsc{Deliver}(h, (O, \pi, e))$

The following theorem establishes that the event-driven compiler realizes the BFT object-log abstraction.

Theorem 1 (DPCM Compiler Theorem). Suppose all correct parties use the same producer slot domain $\mathcal{S}$ and consumer placement algorithm σ. If Π_P satisfies Definition 1 and Π_C satisfies Definition 2, then the compiler $\mathsf{Compile}(\Pi_P, \Pi_C)$ by Algorithm 1 at every party implements the BFT object log of Definition 3.

Proof. *Integrity and total order.* For each height, consumer agreement produces at most one irrevocable decision at a correct party, so Algorithm 1 invokes at most one delivery for that height. Consumer ordered decision emission withholds $\textsc{Decided}(h, d, e)$ until height $h - 1$ has been delivered. The compiler therefore delivers heights in increasing order, even when finalization instances execute out of order.

Agreement. Fix a height h and let $s = \sigma^{-1}(h)$. Consumer agreement ensures that no two correct parties emit different decision bits. If that bit is 0, all correct parties deliver the same dummy entry. If it is 1, producer slot binding fixes one commitment for s, and commitment binding fixes one semantic object. Hence all correct parties that deliver at h deliver the same normal object. Together with integrity and ordered delivery, this per-height agreement makes the correct-party logs prefix-comparable.

External validity. The compiler installs only certificates accepted by $\mathbf{ValidProd}$ and checks $\mathbf{ValidDec}$ before either decision branch. The normal branch additionally verifies $\mathsf{Commit}(O) = c$ after recovery. Hence every normal or dummy entry satisfies the external-validity.

Data availability. Consider the correct party whose input $b = 1$ supports a normal decision. Its producer handler broadcast the valid certificate before making it visible in $cert$. Eventual message delivery therefore exposes a valid certificate for s at every correct party. Producer certified availability then makes $\textsc{Recover}(s, c, \pi)$ return the uniquely committed object, so every correct party can complete the normal entry.

Termination. Consumer coverage eventually emits $\textsc{Ready}(h)$ at every correct party for every fixed height h. Algorithm 1 responds exactly once with $\textsc{Consume}(h, b)$, so consumer termination eventually produces a common decision. A dummy decision is delivered immediately. For a normal decision, binary validity implies that some correct party input 1. Its certificate was broadcast before that input became visible. Producer certified availability therefore makes every asynchronous $\mathsf{CompleteNormal}$ invocation terminate. Ordered decision emission then yields delivery at every fixed height with probability one.

E. Batching in Consumer

The per-height consumer is the semantic interface used by the composition theorem. A concrete system may optimize it by running an agreement instance for a set $\mathcal{H}$ of heights. This *batched consumer* may speculatively resolve multiple heights simultaneously.

A batch instance abides by the following rules:

- A correct party inputs 1 only if it holds a valid producer certificate for every slot $\sigma^{-1}(h)$ with $h \in \mathcal{H}$.
- A decision 1 finalizes every height in $\mathcal{H}$ as normal. A decision 0 finalizes a dummy only when $|\mathcal{H}| = 1$.
- If $|\mathcal{H}| > 1$, a decision 0 finalizes no height. Instead, it deterministically partitions $\mathcal{H}$ into disjoint nonempty child sets, and invokes a new instance for each child.

A batch instance has a public identifier that binds the session and its exact height set $\mathcal{H}$. Its evidence predicate $\mathbf{ValidBatchDec}(\mathcal{H}, d, e)$ is sound and nonconflicting. For $d = 1$, valid batch evidence is accepted as $\mathbf{ValidDec}(h, 1, e)$ for every $h \in \mathcal{H}$. For $d = 0$, evidence is accepted as a per-height dummy decision only when $\mathcal{H} = \{h\}$. A child identifier additionally binds valid decision-0 evidence for its parent. Thus a non-singleton 0 authorizes refinement rather than a log decision. Once a batch resolves heights, the implementation exposes their per-height outcomes in increasing order.

Lemma 1 (Batching in Consumer). Suppose every unresolved height is eventually covered by a batch instance invoked by all correct parties, recursive partitioning is finite, the evidence rules above hold, and resolved heights are exposed in increasing order. Replacing per-height consumer instances

by the batched consumer above preserves the conclusion of Theorem 1.

Proof. Fix a covered height h. Deterministic partitioning into disjoint children gives h a unique chain of containing instances, and agreement makes the decisions along this chain common to all correct parties. Every non-singleton decision 0 advances the chain. Hence termination and finite partitioning ensure that it ends either with decision 1 for a set containing h or with decision 0 for the singleton $\{h\}$. In the first case, some correct party must have input 1, since otherwise binary validity would force decision 0. By the batch-input rule, that party holds a valid producer certificate for every height in the decided set, including h. Producer binding and recovery therefore give h a unique available normal object. In the second case, only the singleton decision is exposed, yielding one dummy resolution at h. Valid, nonconflicting decision evidence makes each resolution externally valid, and ordered delivery preserves total order. Thus the agreement, availability, and termination arguments of Theorem 1 continue to hold for every height.

F. Application Interpretations

BFT SMR [12], [20]. A normal object O may represent an executable transaction block, while a dummy entry represents a deterministic no-op. If all correct parties execute delivered entries in height order using the same deterministic transition function, agreement and total order make them traverse the same sequence of states. Invalid commands may be excluded by the producer verifier or interpreted as deterministic no-ops. Section IV realizes this interpretation in Octopus, using CBC to certify transaction blocks and a batched consumer to finalize their assigned log heights.

BFT storage [16]. Alternatively, O may be authenticated storage metadata for data dispersed outside the consumer. A producer can use AVID-M to store one authenticated coded fragment at each party and certify the resulting Merkle root, while the consumer orders only this certified metadata. DPCM then provides the ordered storage log. Payload retrieval, fragment repair, retention, and access control remain outside the consumer. Appendix D gives this construction and its read path.

Other potential interpretations. The same separation may be useful in systems where independently produced objects must be certified before receiving a common order. A sharded ledger [23] could treat shard batches or cross-shard receipts as producer objects. A randomness beacon could derive its output from cryptographic commitments delivered by the log, following the producer–consumer separation used by AsyRand [29]. A modular blockchain could order authenticated metadata for transaction data whose availability is established separately [2].

IV. Octopus: DPCM-based BFT SMR

This section instantiates the DPCM compiler as an asynchronous BFT SMR protocol–Octopus.

A. Overview

Octopus instantiates the DPCM producer Π_P with parallel consistent broadcast (CBC) instances and the consumer Π_C with asynchronous binary agreement (ABA) and batched ABA. CBC instances continuously certify transaction blocks at queue positions, while ABA instances independently finalize the corresponding log heights. Batched ABA optimizes this consumer step by deciding several heights together and refining a batch that decides 0 into smaller instances. The resulting DPCM object log is interpreted as the Octopus SMR ledger.

Producer variables. Producer P_i owns the queue namespace $Ques[i]$, of which every party maintains a local view. Position (i,k) denotes the k-th block proposed by P_i. The producer-facing fields $Ques[i][k].data$ and $Ques[i][k].cert$ store its candidate block and certificate. Each queue has maximum length L, and $space_i$ records its remaining capacity. A CBC instance disseminates block $\mathbf{B}$ and binds (i,k) to $hv = Hash(\mathbf{B})$. A set of $2f+1$ ready signatures forms the ready quorum certificate RQC. Thus RQC instantiates the DPCM producer certificate π, whereas cert is only the message tag used to carry RQC. Concretely, $\mathbf{ValidProd}((i,k), Hash(\mathbf{B}), \mathsf{RQC}) = 1$ iff RQC verifies for $(i,k,Hash(\mathbf{B}))$.

Consumer variables. The consumer assigns queue positions to log heights using the public round-robin placement algorithm $\sigma(i,k) = (k-1)n+i$. Its inverse is $\sigma^{-1}(h) = (i,k)$, where $i = ((h-1) \bmod n)+1$ and $k = \lfloor (h-1)/n \rfloor + 1$. The n positions with a fixed k form a column, and a batch contains ℓ consecutive columns. Initialized to 0, $\mathfrak{h}$ is the length of the delivered log prefix. For each position, the consumer-facing field $Ques[i][k].status \in \{\mathsf{unresolved}, \mathsf{normal}, \mathsf{dummy}\}$ records the resolution, and $Ques[i][k].evidence$ stores its decision evidence. A normal resolution carries evidence e_N from an ABA or batched ABA decision 1. A dummy resolution carries evidence e_D from a terminal single-height ABA decision 0. A non-terminal batch decision 0 produces no ledger entry and instead triggers refinement. A parameter $\zeta = 2/3$ defines the per-column threshold for BatchReady, which is used to trigger a batched ABA instance.

A normal DPCM entry (O, π, e_N) is therefore instantiated as $(\mathbf{B}, \mathsf{RQC}, e_N)$, whereas $(\perp, e_D)$ is a deterministic no-op. Only consumer-finalized entries are appended to the ledger and executed by the SMR layer. Table II summarizes the notation used in Octopus.

To illustrate status evolution in Octopus, we define the local status of a queue position and then show how resolved positions form the ledger. Table III summarizes a queue position before and after consumer resolution.

Figure 2 then depicts the concrete Octopus ledger layout: blocks are placed by rotating through the n producers, and the identifier (i,k) maps to log height $(k-1)n+i$.

B. The Producer Process

In Octopus, each party independently launches the producer process to continuously generate new blocks. CBC (refer to Appendix A) is used as the producer-certified availability

TABLE II
THE NOTATIONS USED IN Octopus

Variable	Description
$\mathbf{B}$	Executable transaction block
(i,k)	The k-th block proposed by party P_i
$Ques[i]$	Local view of producer P_i's queue; $Ques[i][k]$ stores position (i,k)
L	Maximum length of each producer queue
$space_i$	Remaining capacity of producer queue $Ques[i]$
hv	Block digest, i.e., $hv = Hash(\mathbf{B})$
RQC	CBC ready quorum certificate binding (i,k) to hv
cert	Message tag used to disseminate an RQC
σ	Placement algorithm $\sigma(i,k) = (k-1)n + i$
$\mathfrak{h}$	Length of the delivered ledger prefix
ℓ	Number of epochs/columns in a batch
$\zeta = 2/3$	Per-column threshold used by BATCHREADY
e_N	ABA or batched ABA evidence resolving covered positions as normal with decision bit 1
e_D	Single-height ABA evidence finalizing one position as dummy with decision bit 0

TABLE III
EXPLANATIONS OF DIFFERENT BLOCK STATUS

unresolved	i,k	Unresolved queue position
	i,k	Producer-certified candidate block $\mathbf{B}$ with RQC
normal	i,k	Normal entry containing $\mathbf{B}$ with e_N
dummy	i,k	Dummy entry $\perp$ with e_D

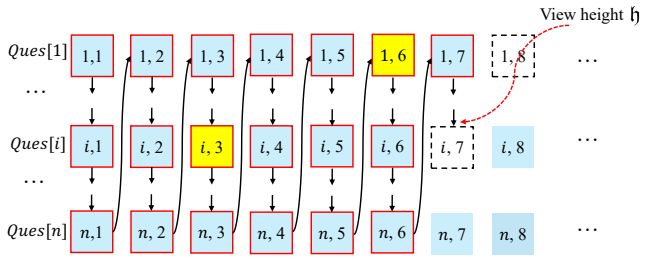


Fig. 2. The Octopus ledger layout where blocks are linked by sequentially rotating all the n parties

mechanism: a proposer broadcasts an initial message carrying $\mathbf{B}$, parties return ready signatures over $(i,k,Hash(\mathbf{B}))$, and any $2f+1$ such signatures form a RQC for position (i,k).

Algorithm 2 gives the producer process. If proposer i has available queue space, it generates a block $\mathbf{B}$ and broadcasts $(\mathsf{initial}, v)$ with $v = (\mathbf{B}, i, k)$ (lines 1–4). Upon receiving $(\mathsf{initial}, v)$, party j stores the block and sends a ready vote $(\mathsf{ready}, hv, i, k)$, where $hv = Hash(\mathbf{B})$ (lines 5–7). Once $2f+1$ ready votes for the same statement are collected, the producer forms RQC and broadcasts it in a cert message (lines 8–10). A party relays a previously unseen valid RQC before installing it, invokes $\mathsf{pull}(i,k)$ if the block is missing,

Algorithm 2 The producer process

1: **upon** $space_i > 0$ at proposer i
2: generates a new block $\mathbf{B}$ as PRODUCE event
3: $space_i \leftarrow space_i - 1$
4: broadcasts $(\mathsf{initial}, v)$, where $v = (\mathbf{B}, i, k)$.
5: **upon** receipt of $(\mathsf{initial}, v)$ at party j
6: store $\mathbf{B}$ for (i,k,hv), where $hv = Hash(\mathbf{B})$
7: sends $(\mathsf{ready}, hv, i, k)$ to i.
8: **upon** receipt of $2f+1$ $(\mathsf{ready}, hv, i, k)$ at party j
9: RQC $\leftarrow$ the collected ready signatures
10: broadcasts $(\mathsf{cert}, \mathsf{RQC}, hv, i, k)$
11: **upon** receipt of $(\mathsf{cert}, \mathsf{RQC}, hv, i, k)$ at party j
12: **if** $Ques[i][k].cert = null$ and RQC verifies for (i,k,hv) **then**
13: broadcast $(\mathsf{cert}, \mathsf{RQC}, hv, i, k)$ as CERTIFIED event
14: $Ques[i][k].cert \leftarrow \mathsf{RQC}$
15: $Ques[i][k].data \leftarrow \mathbf{B}$ if local $\mathbf{B}$ has hash hv; otherwise $\mathsf{pull}(i,k)$
16: $space_i = space_i + 1$

and releases one queue position (lines 11–16).

CBC does not provide totality. One correct party may hold a valid RQC and its block while another lacks either. Upon receiving a valid RQC, party j stores a matching local $\mathbf{B}$ or invokes $\mathsf{pull}(i,k)$ otherwise (line 15). Recovery does not delay the consumer. The requester samples without replacement from the $2f+1$ ready signers identified by RQC and contacts another signer at a later retry step if necessary. At most f signers are Byzantine. Hence the first $f+1$ distinct requests include a correct signer that stored a block matching hv, and eventual delivery completes recovery. Algorithm 3 gives the responder logic. A party returns $(\mathsf{RQC}, \mathbf{B})$ when both are locally available.

Algorithm 3 Service for pull request at each party

1: **upon** $\mathsf{pull}(i,k)$ from any other party
2: $\mathbf{B} \leftarrow Ques[i][k].data$
3: **if** $Ques[i][k].cert \neq null$ and $\mathbf{B} \neq \perp$ **then**
4: respond with $(Ques[i][k].cert, \mathbf{B})$.

Figure 3 illustrates an example from the producer perspective at party j. Different parties, as producers, may generate blocks at varying heights, reflecting asynchronous progress across the network. In party j's view, block $(i,6)$ from party i and block $(n,8)$ from party n remain without stored certificates, while the other produced blocks have stored valid RQC certificates under the CBC protocol.

C. The Consumer Process

The consumer uses ABA to resolve one scheduled height and ABA_{batch} to apply one binary decision to a finite height set $\mathcal{H}$. The latter is a batched instance, not a vector of per-height decisions. Appendix B recalls their common ABA interface. A party inputs 1 exactly when it holds a valid RQC for every

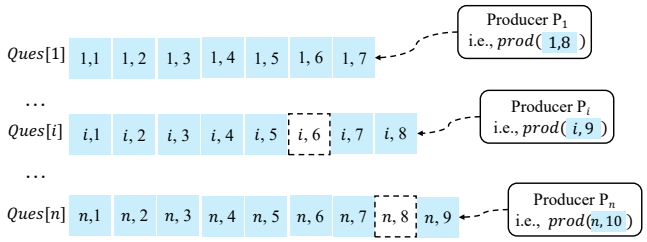


Fig. 3. Producer view at party j.

covered position and inputs 0 otherwise. A decision 1 finalizes all covered positions as normal after their certificates and blocks are verified. A decision 0 finalizes a dummy only in a single-height ABA instance, while a non-singleton ABA_{batch} decision 0 triggers refinement.

An output (d, e) contains the decision bit d and its publicly verifiable decision evidence e, which Octopus stores without inspecting its internal structure. Deterministic instance identifiers ensure that correct parties satisfying the same invocation condition join the same ABA or batched ABA instance. The following optimistic and pessimistic paths specify how these instances cover and refine height sets.

Algorithm 4 Shared consumer helpers at party j

```
1:  func IsAllRQC(H)
2:      for all h ∈ H do
3:          (i, k) ← σ^{-1}(h)
4:          if Ques[i][k].cert is not a valid RQC then
5:              return false
6:      return true
7:  func FinalizeNormal(H, e)
8:      for all h ∈ H do
9:          (i, k) ← σ^{-1}(h)
10:         obtain and verify (B, RQC) for (i, k), invoking pull if needed
11:         Ques[i][k].data ← B
12:         Ques[i][k].evidence ← e
13:         Ques[i][k].status ← normal
14: func AdvanceView()
15:     while Ques[σ^{-1}(ɦ + 1)].status ≠ unresolved do
16:         deliver the resolved entry at h = ɦ + 1;
17:         ɦ ← h
```

Both paths use the local operations in Algorithm 4. IsAllRQC tests whether every height in a set has a valid producer certificate. FinalizeNormal verifies or recovers the corresponding blocks and records their decision evidence, while AdvanceView exposes only the longest consecutive prefix whose positions have been resolved.

1) Optimistic Path: The optimistic path amortizes agreement across ℓ consecutive producer columns. When every height in a batch has a valid RQC, the consumer submits 1 to one ABA_{batch} instance rather than invoking a separate ABA for each height. A decision 1 finalizes the entire batch as normal. Implementations may invoke different batches early and overlap their execution, while AdvanceView exposes their finalized entries only in height order. Figure 4 illustrates two consecutive optimistic batches.

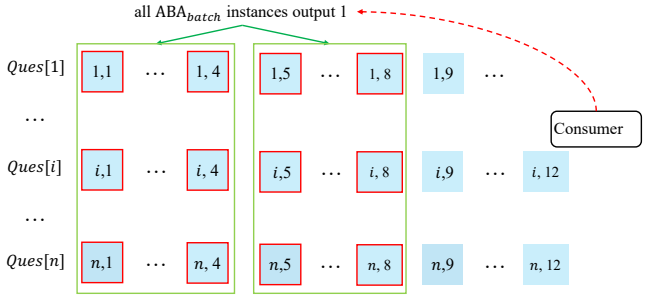


Fig. 4. An optimistic execution of the consumer with batch size $\ell = 4$. The first batch covers columns 1–4 and advances ɦ from 0 to $4n$. The second covers columns 5–8 and advances it to $8n$. Under early invocation, the two agreement instances may overlap, but their entries become visible in log order.

Waiting for every certificate in the current batch would let a faulty producer block its invocation. Let the current ℓ-column batch span $h_0 = (k_0 - 1)n + 1$ through $h_1 = (k_0 + \ell - 1)n$. Denote $\mathcal{C}_r$ as the batch's r look-ahead columns, i.e., $\mathcal{C}_r = [h_1 + (r-1)n + 1, h_1 + rn]$. Algorithm 5 raises BATCHREADY when one of the two conditions are met: (i) the current batch is fully certified; (ii) every column in $\mathcal{C}_r$ contains at least $\lceil 2n/3 \rceil$ certified queue positions. The latter condition can be satisfied since $n - f > 2n/3$ and it is leveraged in pessimistic path to ensure progress.

Algorithm 5 Optimistic consumer path at party j

```
1: event BATCHREADY(h_0, h_1) is emitted if
       (i) IsAllRQC([h_0, h_1]) holds, or
       (ii) every C_r, r ∈ [1, ℓ], has at least ⌈ζn⌉ = ⌈2n/3⌉ valid certificates.
2: upon BATCHREADY(h_0, h_1) as READY event
3:     H ← [h_0, h_1]
4:     b ← IsAllRQC(H)
5:     init ABA_batch(h_0, h_1, b) as CONSUME event
6: upon ABA_batch outputs (1, e) for batch [h_0, h_1]
7:     FinalizeNormal([h_0, h_1], e) as DECIDED event
8:     AdvanceView() as DELIVER event
```

Upon BATCHREADY, the party recomputes IsAllRQC($[h_0, h_1]$) and starts or joins the batch instance with the resulting bit (lines 2–5). Condition (i) yields input 1. Condition (ii) may yield input 0 and initiate refinement without further waiting. An output $(1, e)$ invokes FinalizeNormal to verify or recover every covered block and record e. Then AdvanceView delivers the longest consecutive resolved prefix (lines 6–8). An output 0 finalizes nothing at this stage and enters the pessimistic path below.

2) Pessimistic Path: The pessimistic path resolves a current batch after its ABA_{batch} instance outputs 0. A batch output 0 does not assign 0 to every covered height and therefore produces no dummy entry. Instead, the consumer partitions the batch by producer row and recursively refines only the row

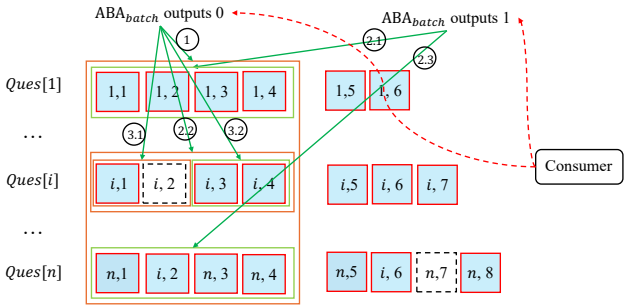
segments that continue to output 0. A row segment that outputs 1 is resolved as normal, whereas an output 0 yields a dummy resolution only at a singleton. Row and child instances are independent and may execute in parallel. Figure 5 illustrates a pessimistic execution.

Algorithm 6 implements this procedure. For producer i and columns k_a through k_b, let $\mathcal{H}_i[k_a, k_b] = \{\sigma(i,k) \mid k_a \le k \le k_b\}$ denote the row segment. After the initial batch outputs 0, the first handler maps h_0 and h_1 through σ^{-1} to obtain the complete column range k_0 through k_1, then calls StartSegment for all producer rows in parallel (lines 1–4). StartSegment derives the local input using IsAllRQC and invokes ABA_{batch} for a multi-block segment or ABA for a singleton (lines 21–25).

For a row segment, output 1 invokes FinalizeNormal, whereas output 0 invokes RefineSegment (lines 5–9), which bisects the column range and starts both children (lines 26–29). Each step strictly shortens the unresolved row segments. Every refinement branch therefore reaches either a segment decided as normal or a singleton. At a singleton, output 1 records a normal entry and output 0 records a dummy entry with evidence $e_D = e$ (lines 11–17). After either resolution, AdvanceView delivers only the longest consecutive resolved prefix (lines 10 and 18). This preserves log order even though the refinement instances execute concurrently.

D. Security Proof via DPCM

Theorem 2 (Correctness of Octopus SMR). Under the system model of Section III-B, assume collision-resistant hashing and unforgeable signatures. Correct parties sign at most one ready statement per queue position and retain the matching block. Each correct producer supplies a valid block, possibly empty, and continues proposing at successive available positions using CBC. Every ABA and ABA_{batch} instance satisfies agreement, termination, binary validity, and decision-evidence soundness. Under these assumptions, Octopus implements an asynchronous BFT SMR protocol.



Unhappy case illustrating hybrid invocation of ABA instances: (1) for columns from 1 to 4; (2.1) for row 1 and columns from 1 to 4; (2.2) for row *i* and columns from 1 to 4; (2.3) for row *n* and columns from 1 to 4; (3.1) for row *i* and columns from 1 to 2; (3.2) for row *i* and columns from 3 to 4 ……

Fig. 5. Example of pessimistic execution of the consumer. Rows are tested in parallel. A decision 1 resolves a segment as normal, whereas a decision 0 triggers recursive splitting and yields a dummy only at a singleton.

Algorithm 6 Pessimistic consumer path at party j

1: **upon** ABA_{batch} outputs $(0, e)$ for batch $[h_0, h_1]$:
2: $(1, k_0) \leftarrow \sigma^{-1}(h_0)$ and $(n, k_1) \leftarrow \sigma^{-1}(h_1)$
3: **for all** $i \in [1, n]$ **in parallel do**
4: StartSegment(i, k_0, k_1)
5: **upon** ABA_{batch} outputs (d, e) for row segment $\mathcal{H}_i[k_a, k_b]$:
6: **if** $d = 1$ **then**
7: FinalizeNormal$(\mathcal{H}_i[k_a, k_b], e)$
8: **else**
9: RefineSegment(i, k_a, k_b)
10: AdvanceView()
11: **upon** ABA outputs (d, e) as DECIDED event for the block at $h = \sigma(i, k)$:
12: **if** $d = 1$ **then**
13: FinalizeNormal$(\{h\}, e)$
14: **else**
15: $Ques[i][k].data \leftarrow \perp$
16: $Ques[i][k].evidence \leftarrow e$
17: $Ques[i][k].status \leftarrow$ dummy
18: AdvanceView()
19: **func** StartSegment(i, k_a, k_b)
20: $\mathcal{H} \leftarrow \mathcal{H}_i[k_a, k_b]$
21: $b \leftarrow$ IsAllRQC$(\mathcal{H})$
22: **if** $k_a \equiv k_b$ **then**
23: init ABA$(\sigma(i, k_a), b)$
24: **else**
25: init $\mathsf{ABA}_{batch}(\mathcal{H}, b)$
26: **func** RefineSegment(i, k_a, k_b)
27: $k_m \leftarrow \lfloor (k_a + k_b)/2 \rfloor$
28: StartSegment(i, k_a, k_m)
29: StartSegment$(i, k_m + 1, k_b)$

Proof. *Producer instantiation.* For slot $s = (i, k)$, Π_P maps PRODUCE$(s, \mathbf{B})$ to the CBC proposal. It emits CERTIFIED$(s, Hash(\mathbf{B}), \mathsf{RQC})$ when a valid RQC is installed and implements RECOVER by local lookup and pull(i, k). The predicate **ValidProd** verifies the $2f + 1$ ready signatures on (i, k, hv). Two valid certificates for the same slot have at least $f + 1$ common signers, including a correct party. Since a correct party signs at most one digest per slot, both certificates bind the same hv. Collision resistance then binds the corresponding block. Every valid RQC also contains at least $f+1$ correct signers that stored the matching block before signing. Certificate relay and recovery from distinct signers therefore give verifiability, slot binding, and availability as required by Definition 1.

Consumer instantiation. The round-robin $\sigma(i, k) = (k - 1)n + i$ is a bijection from producer slots to log heights. A single-height ABA directly realizes binary height finalization. For a finite set $\mathcal{H}$, ABA_{batch} realizes the extension of Lemma 1. A correct party inputs 1 only when IsAllRQC$(\mathcal{H})$ verifies a certificate for every covered slot. Decision 1 supplies valid normal evidence for each $h \in \mathcal{H}$. Decision 0 produces

a log outcome only for a singleton. A non-singleton decision 0 instead authorizes the deterministic child instances. Their identifiers are determined by the public session and covered height sets, so Byzantine messages cannot select a different refinement at a correct party. The underlying ABA instances process only height sets, bits, and decision evidence. Agreement and deterministic refinement give all correct parties the same terminal resolution for each height. By binary validity and the batch-input rule, every decision 1 is supported by valid certificates for all covered slots. Only a singleton decision 0 becomes a dummy, and decision-evidence soundness authenticates either outcome. The compiler-side FinalizeNormal obtains and verifies each normal block. The operation AdvanceView exposes resolved entries only along the consecutive prefix, leaving payload recovery outside agreement. Lemma 5 in Appendix C establishes eventual root-batch invocation, finite refinement, and completion of every normal resolution. Hence the batched consumer satisfies the coverage and progress conditions of Lemma 1.

Composition. Theorem 1 and Lemma 1 now give a BFT object log. If all correct parties start from the same state and apply the same deterministic validation and transition function, normal entries execute transaction blocks and dummy entries execute no-ops, so their replicated states remain identical after every delivered height.

V. Evaluations

A. Complexity Analysis

We normalize communication and message complexity to one column of n blocks, comparable to one ACS-style output. The consumer operates on batches of ℓ columns, so its batch cost is amortized over ℓ. Here $n = 3f+1$, f is the fault bound, λ is the security parameter, and $|\mathbf{B}|$ is the average block size. For a phase X, M_X and C_X denote its point-to-point message and bit complexities, respectively. We assume compact $O(\lambda)$-bit quorum certificates. Consumer latency is measured after BatchReady. CBC and recovery proceed asynchronously outside the consumer critical path. Following the CBC analysis of ALEA-BFT [4], one CBC instance costs $O(n)$ messages, $O(n(|\mathbf{B}| + \lambda))$ bits, and expected $O(1)$ asynchronous time. One ABA or ABA_{batch} instance costs expected $O(n^2)$ messages, $O(\lambda n^2)$ bits, and expected $O(1)$ time. It has shown that completing all n instances takes expected $O(\log n)$ rounds [5].

Producer and recovery. The n CBC instances in one column cost $O(n^2)$ messages and $O(n^2(|\mathbf{B}| + \lambda))$ bits. For a certified queue position s, let q_s be the number of correct parties that have its RQC but lack the matching block, and let $r = \sum_s q_s$ over the column. Since the $2f + 1$ ready signers include at least $f + 1$ correct block holders, $q_s \le f$ and $r \le fn$. A requester probes distinct signers in uniformly random order. If T is the number of probes through the first correct holder, then $\mathbb{E}[T] \le \frac{2f+2}{f+2} < 2$. Each probe has constant message cost, and the successful response carries one certificate and block. Thus the expected producer and recovery costs per column are $\mathbb{E}[M_P(r)] = O\big(n^2 + r\big), \mathbb{E}[C_P(r)] = O\big(n^2(|\mathbf{B}| + \lambda) + r(|\mathbf{B}| + \lambda)\big)$. Because $r \le fn$, these totals remain $O(n^2)$ messages and $O(n^2(|\mathbf{B}| + \lambda))$ bits even when every certified queue position is missing at f correct parties. This expectation counts probes through the first correct holder. Under an adversarial schedule, a requester may contact up to $f + 1$ signers before receiving a valid response, giving conservative schedule-independent bounds of $O(n^3)$ messages and $O(n^3(|\mathbf{B}| + \lambda))$ bits per column.

Optimistic consumer. If the initial ABA_{batch} decides 1, one instance finalizes all ℓn positions. Its $O(n^2)$ messages and $O(\lambda n^2)$ bits per batch amortize to $O(n^2/\ell)$ messages and $O(\lambda n^2/\ell)$ bits per column. Producer work dominates, giving the expected per-column total $M_{\mathsf{opt}} = O(n^2), C_{\mathsf{opt}} = O(n^2(|\mathbf{B}| + \lambda))$ and an expected $O(1)$ consumer critical path.

Pessimistic consumer. A decision 0 for the initial batch invokes n row instances in parallel. If $c \le f$ Byzantine-producer rows require complete refinement while the other rows stop at their roots, each refined ℓ-position row contributes a binary tree of at most $2\ell - 1$ instances. Counting the initial batch and the n row roots gives $N_{\mathsf{ABA}} \le 1 + n + c(2\ell - 2) = O(n + c\ell)$. The consumer therefore costs $O((n + c\ell)n^2)$ messages and $O(\lambda(n + c\ell)n^2)$ bits per batch, and $O\left(\frac{n^3}{\ell} + cn^2\right)$ messages, $O\left(\frac{\lambda n^3}{\ell} + \lambda cn^2\right)$ bits per column. Including producer work and expected randomized recovery, the total per-column costs are $O\left(n^2 + \frac{n^3}{\ell} + cn^2\right)$ messages and $O\left(n^2(|\mathbf{B}| + \lambda) + \frac{\lambda n^3}{\ell} + \lambda cn^2\right)$ bits. The two consumer terms separate the effects of ℓ and c. Increasing ℓ amortizes the initial and n row-root instances, reducing the first term as n^3/ℓ. In contrast, each of the c fully refined rows contains $\Theta(\ell)$ instances per batch and therefore contributes $\Theta(n^2)$ per column, making the second term linear in c but independent of ℓ. Thus the cost is $O(n^3/\ell)$ when $c = 0$. For $c > 0$, the refined-row term dominates once $\ell = \Omega(n/c)$. At the extreme $c = f = \Theta(n)$, the per-column cost is $O(n^3)$ regardless of further increases in ℓ. The n row-root instances execute concurrently, but completing the batch requires expected $O(\log n)$ rounds. A fully refined row adds at most $\lceil \log_2 \ell \rceil$ causally dependent bisection levels. Hence pessimistic consumer latency is expected $O(\log n + \log(\ell + 1))$. The parameter c affects total work but not this asymptotic critical path because rows and sibling instances run concurrently. At most n rows can refine, including delayed honest rows. Bounding each refined row by a full tree preserves the $O(n^3)$ worst-case work bound.

B. Experimental Results

a) Implementation: We implement Octopus, Dumbo-NG [18], and Alea-BFT [4] in Go within a common experimental framework. All three implementations use the same TCP networking code, cryptographic libraries, transaction generator, block format, and measurement code. In particular, they use Ed25519 to authenticate protocol messages and the same threshold-signature implementation where required. Octopus implements the CBC introduced in Appendix A and Cobalt ABA [25] with a threshold-signature common coin, as does Alea-BFT [4]. This setup keeps the shared system

components and measurement method identical while allowing each protocol to retain its own broadcast and agreement logic.

b) Experimental setup: Our WAN deployment spans eight Alibaba Cloud regions: Virginia, Silicon Valley, Frankfurt, Dubai, Tokyo, Hong Kong, Singapore, and Jakarta. Parties are assigned cyclically across these regions, and each active party runs on a dedicated ECS `ecs.c9i.large` instance with 2 vCPUs, 4 GiB of memory, and 100 Mbps public egress. We evaluate $n \in \{8, 16, 32\}$ parties. For each value of n, we test both $f = 0$ and the corresponding maximum number $f_{\max} \in \{2, 5, 10\}$ of silent faulty parties. We model a silent fault by not starting the corresponding party. Transactions are 250 bytes, and each block contains 200, 600, 1000, 1400, 1800, or 2200 transactions. For all protocols, latency is measured from block broadcast until ordered delivery into the local gap-free log prefix; in Octopus, this is the DELIVER event emitted by AdvanceView. Throughput is delivered normal-block transactions divided by the elapsed time from the first broadcast to the last delivery; dummy entries contribute no transactions.

c) Parameter selection: We evaluate 72 parameter settings using the same WAN deployment and 500-transaction blocks: $n \in \{8, 16, 32\}$, $f \in \{0, f_{\max}\}$, $\ell \in \{1, 2, 5, 10\}$, and $L \in \{2\ell, 3\ell, 4\ell\}$. The complete results appear in Figure 8 in Appendix E. They show that increasing ℓ or L generally improves throughput at the cost of higher latency, with the trade-off becoming more pronounced as n increases.

TABLE IV
SELECTED Octopus PARAMETERS AND THEIR PERFORMANCE WITH 500-TRANSACTION BLOCKS. EACH PERFORMANCE ENTRY REPORTS $f = 0/f_{\max}$.

n	ℓ	L	Throughput (kTPS)	Latency (s)
8	2	8	20.7/10.9	1.25/2.05
16	1	4	15.2/11.4	1.85/1.87
32	1	3	8.2/7.7	5.31/4.14

No single setting maximizes throughput and minimizes latency under both fault settings. We therefore select one setting for each value of n by jointly considering throughput and latency under $f = 0$ and $f = f_{\max}$. Table IV reports the selected settings and their performance. In particular, we use $\ell = 1$ at $n = 16$ and $n = 32$ to avoid the substantial latency increase caused by larger batches.

d) WAN performance: The upper row of Figure 6 reports WAN throughput as the number of transactions per block increases. With 2200 transactions per block and no faults, Octopus reaches 95.4%, 93.5%, and 93.5% of Dumbo-NG's throughput at $n = 8, 16, 32$, respectively. With the maximum number of silent faults, Octopus exceeds Dumbo-NG by 25.9% at $n = 8$ and 18.7% at $n = 16$, while Octopus and Dumbo-NG differ by less than 1% at $n = 32$. Thus, for large blocks, Octopus achieves throughput comparable to Dumbo-NG without faults and matches or exceeds Dumbo-NG with silent faults. At the same block size, Alea-BFT achieves 9.9/3.8, 10.5/2.1, and 10.7/1.4 kTPS at $n = 8, 16, 32$ under $f = 0/f_{\max}$, respectively. Across the six evaluated (n, f) settings, Octopus achieves 2.9–16.8× the throughput of Alea-BFT.

The throughput gap between Octopus and Dumbo-NG narrows as larger blocks amortize Octopus's fixed per-block CBC, signature, and ABA work. More importantly, Octopus controls network load through producer space. Ordered delivery releases this space, and a producer stops starting new CBC instances when its queue reaches the maximum length L. The producer-space bound supplies the backpressure, while CBC pacing smooths each admitted payload broadcast. Dumbo-NG has no corresponding bound between broadcast and agreement. Its producers continue broadcasting while MVBA instances advance sequentially, so payload and MVBA messages can create bursts on the same 100 Mbps egress and trigger TCP retransmissions. At $n = 32, f = 0$ and 2200 transactions, Dumbo-NG's host-reported p95/peak transmit rates are 153/306 Mbps, compared with 79/90 Mbps for Octopus. This traffic control explains why Octopus records fewer TCP retransmissions and gains more from larger blocks than Dumbo-NG.

The lower row of Figure 6 reports latency. At $n = 8$, Octopus and Dumbo-NG remain in a comparable 1.1–2.7 s range. With 2200 transactions per block, Dumbo-NG's latency is 2.49/1.93× that of Octopus at $n = 16$ under $f = 0/f_{\max}$, and 1.93/2.06× at $n = 32$. At the same block size, Alea-BFT's latency is 5.38/13.21, 8.21/39.62, and 9.97/86.30 s at $n = 8, 16, 32$ under $f = 0/f_{\max}$, respectively. Across the six evaluated (n, f) settings, Alea-BFT's latency is 2.4–15.2× that of Octopus.

At $n = 32, f = 0$, increasing the block size from 200 to 1800 transactions reduces Octopus's latency from 5.49 to 4.65 s. Although broadcast time rises from 1.70 to 2.07 s, agreement-start-to-delivery time falls from 2.02 to 0.80 s. Larger blocks slow producer progress, so a root batch is more likely to have all RQCs when triggered and take the optimistic path. Accordingly, root decisions of 0 fall from 31 to 20 and the number of row instances falls from 992 to 640. With the maximum tested number of silent faults, missing RQCs disable the optimistic path. Nevertheless, at $n = 32, f = 10$ and 2200 transactions, broadcast takes about 1.91 s for both Octopus and Dumbo-NG, whereas wait and agreement-to-delivery take 2.87/0.91 s in Octopus and 5.02/4.84 s in Dumbo-NG. The latency gap between Octopus and Dumbo-NG therefore appears after broadcast. Octopus allows consumer batches to overlap and runs row instances concurrently, while the maximum producer-queue length L bounds concurrent CBC instances. Dumbo-NG advances MVBA instances sequentially.

e) Summary: The selected values of ℓ and L balance throughput and latency under both evaluated fault settings. Larger blocks amortize fixed work. Without faults, they also make the optimistic path more common and avoid many row instances. With maximum faults, the optimistic path is unavailable. Octopus instead overlaps consumer batches, runs row instances concurrently, and bounds concurrent CBC

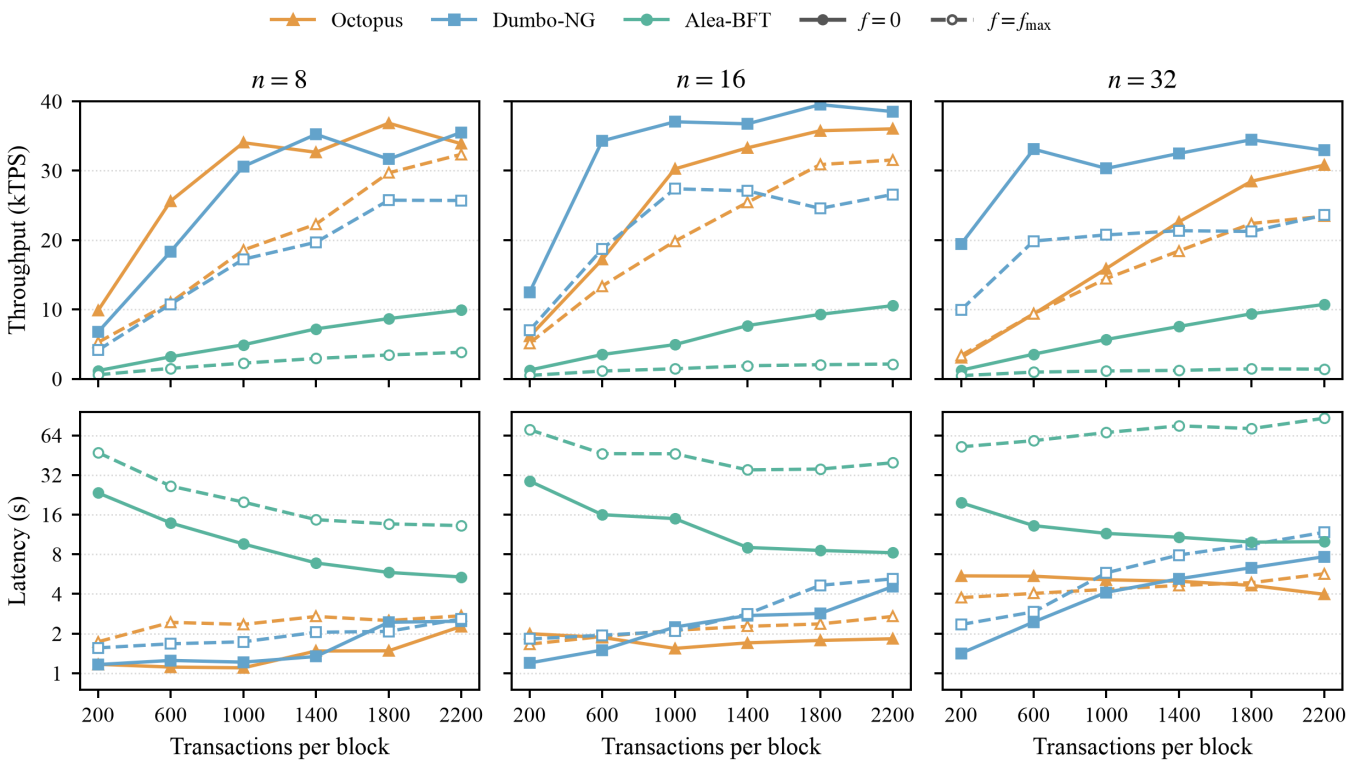


Fig. 6. Alibaba Cloud WAN throughput (upper row) and latency (lower row) as the number of transactions per block varies. Columns show $n \in \{8, 16, 32\}$, and each panel contains the $f = 0$ and $f = f_{\max} = n/3+1$ curves of all three protocols. Solid lines with filled markers denote $f = 0$, while dashed lines with hollow markers denote $f = f_{\max}$. The latency axes share a base-2 logarithmic scale. Transactions are 250 bytes. Octopus uses $(\ell, L) = (2, 8), (1, 4), (1, 3)$ for $n = 8, 16, 32$.

instances through L, whereas Dumbo-NG advances MVBA instances sequentially. Overall, for large blocks, Octopus provides throughput comparable to or higher than that of Dumbo-NG; at $n = 16$ and $n = 32$, Dumbo-NG's latency is 1.9–2.5× that of Octopus. Octopus achieves 2.9–16.8× Alea-BFT's throughput, while Alea-BFT's latency is 2.4–15.2× that of Octopus.

VI. Conclusion

This paper presented DPCM, an asynchronous BFT object-log abstraction that separates payload-specific certified availability from payload-oblivious binary finalization. Its composition theorem identifies the producer and consumer guarantees sufficient for every predetermined log height to finalize as a recoverable normal entry or an evidence-backed dummy. The batched consumer preserves these per-height semantics while allowing one ABA decision to finalize multiple positions. When an optimistic batch fails, deterministic concurrent refinement isolates unresolved positions without placing payload recovery inside agreement. We instantiated DPCM as Octopus, combining continuous CBC producers with a batched ABA consumer. The construction keeps production, agreement, and recovery concurrent and provides timeout-free log progress. Our parameter study shows that increasing the consumer batch size or producer-queue length can improve throughput at the cost of higher latency, especially as the number of parties increases. Across the evaluated eight-region WAN, with 2200-transaction blocks, Octopus achieves throughput comparable to Dumbo-NG without faults and matches or outperforms it with the maximum tested number of silent faulty parties. At $n = 16$ and $n = 32$, Dumbo-NG's commit latency is 1.9–2.5× that of Octopus.

The appendix presents Octopus^{VID}, a BFT storage-log construction that illustrates an alternative producer interface. A complete BFT storage system with retention, repair, garbage collection, and access policies remains future work. We will also explore the other applications from Section III-F based on DPCM, including sharded ledgers, randomness beacons, and modular blockchains. More broadly, DPCM's event-driven producer and consumer interfaces position it as a programmable middleware layer. We believe this abstraction can facilitate the construction of modular distributed systems.

References

[1] I. Abraham, D. Malkhi, and A. Spiegelman. Asymptotically optimal validated asynchronous Byzantine agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, 2019.

[2] M. Al-Bassam. LazyLedger: A distributed data availability ledger with client-side smart contracts. *arXiv preprint arXiv:1905.09274*, 2019.

[3] N. Alhaddad, S. Das, S. Duan, L. Ren, M. Varia, Z. Xiang, and H. Zhang. Balanced Byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 399–417, 2022.

[4] D. S. Antunes, A. N. Oliveira, A. Breda, M. G. Franco, H. Moniz, and R. Rodrigues. Alea-BFT: Practical asynchronous Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2024.

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

[6] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 31–45, 2011.

[7] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

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

[9] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2005.

[10] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 115–124, 2006.

[11] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

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

[13] H. Cheng, Y. Lu, Z. Lu, Q. Tang, Y. Zhang, and Z. Zhang. JUMBO: Fully asynchronous BFT consensus made truly scalable. *IEEE Transactions on Dependable and Secure Computing*, 22(5):5584–5602, 2025.

[14] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 34–50, 2022.

[15] X. Dai, Y. Yu, S. Duan, R. Hao, J. Xiao, and H. Jin. Icarus: Achieving performant asynchronous BFT with only optimistic paths. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2026.

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

[17] 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.

[18] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Dumbo-NG: Fast asynchronous BFT consensus with throughput-oblivious latency. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1187–1201, 2022.

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

[20] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Dumbo: Faster asynchronous BFT protocols. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 803–818, 2020.

[21] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Speeding Dumbo: Pushing asynchronous BFT closer to practice. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022.

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

[23] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 583–598, 2018.

[24] Y. Lu, Z. Lu, Q. Tang, and G. Wang. Dumbo-MVBA: Optimal multi-valued validated asynchronous Byzantine agreement, revisited. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2020.

[25] E. MacBrough. Cobalt: BFT governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

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

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

[28] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse. DispersedLedger: High-throughput Byzantine consensus on variable bandwidth networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 493–512, 2022.

[29] L. Zhang, T. Liu, Z. Ou, H. Kan, and J. Zhang. AsyRand: Fast asynchronous distributed randomness beacon with reconfiguration. *IACR Cryptology ePrint Archive*, Report 2025/406, 2025.

[30] I. Abraham, N. Ben-David, and S. Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2022.

[31] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

[32] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

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

[34] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993.

[35] S. Das, Z. Xiang, and L. Ren. Asynchronous data dissemination and its applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2705–2721, 2021.

[36] S. Das, A. Momose, S. Duan, L. Ren, S. Liu, and V. Shoup. Asynchronous consensus without trusted setup or public-key cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[37] S. Duan, X. Wang, and H. Zhang. FIN: Practical signature-free asynchronous common subset in constant time. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 815–829, 2023.

[38] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[39] A. Gagol, D. Lesniak, D. Straszak, and M. Swietek. Aleph: Efficient atomic broadcast in asynchronous networks with Byzantine nodes. In *Proceedings of the ACM Conference on Advances in Financial Technologies (AFT)*, 2019.

[40] F. Gai, J. Niu, I. Beschastnikh, C. Feng, and S. Wang. Scaling blockchain consensus via a robust shared mempool. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2023.

[41] P. Jovanovic, L. Kokoris-Kogias, B. Kumara, A. Sonnino, P. Tennage, and I. Zablotchi. Mahi-Mahi: Low-latency asynchronous BFT DAG-based consensus. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 549–559, 2025.

[42] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. All you need is DAG. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 165–175, 2021.

[43] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 204–215, 2005.

[44] Y. Lu, Z. Lu, and Q. Tang. Bolt-Dumbo Transformer: Asynchronous consensus as fast as the pipelined BFT. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2159–2173, 2022.

[45] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

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

[47] A. Patra, A. Choudhary, and C. P. Rangan. Simple and efficient asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2009.

[48] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[49] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.
[50] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
[51] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2705–2718, 2022.
[52] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.
[53] C. Stathakopoulou, M. Pavlovic, and M. Vukolic. State machine replication scalability made simple. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 17–33, 2022.
[54] R. Vassantlal, H. Heydari, B. Ferreira, and A. Bessani. MVP-ORAM: A wait-free concurrent ORAM for confidential BFT storage. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2026.
[55] H. Zhang and S. Duan. PACE: Fully parallelizable BFT from reproposable Byzantine agreement. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3151–3164, 2022.

APPENDIX A
CONSISTENT BROADCAST (CBC)

Consistent broadcast (CBC) [3], [20] is the producer-side primitive used by Octopus to certify proposer blocks. It provides validity and agreement. If the proposer is correct, all correct parties eventually deliver its proposal. Any two values delivered by correct parties are identical. Unlike reliable broadcast, CBC does not require totality when the proposer is faulty. One correct party may obtain a ready quorum certificate (RQC) while another remains unresolved. The consumer does not restore CBC totality. Instead, it tolerates a missing certificate by eventually resolving the corresponding log height as normal or dummy.

Figure 7 shows the certificate-producing CBC pattern used by the Octopus producer. A proposer broadcasts an initial message carrying a block. After storing and validating the block, a party signs a ready statement $(i, k, H(\mathbf{B}))$ and retains the matching block until the position is resolved or a safe garbage-collection rule applies. Any $2f + 1$ such signatures form an RQC for position (i, k). Different signer subsets may form different RQCs, but quorum intersection and the one-signature-per-slot rule make every valid RQC for a slot bind the same digest.

APPENDIX B
ASYNCHRONOUS BINARY AGREEMENT (ABA)

Asynchronous binary agreement (ABA) allows parties to agree on one bit without assuming bounds on message delay [7], [30], [46]. Each party submits an input $b \in \{0, 1\}$ and eventually obtains an irrevocable decision $d \in \{0, 1\}$. Agreement prevents two correct parties from deciding differently. Binary validity requires unanimous correct input b to force decision $d = b$. Termination requires every participating correct party to decide with probability one. To circumvent the impossibility of deterministic asynchronous consensus [17], the ABA implementations considered here use randomization, typically through an asynchronous common coin.

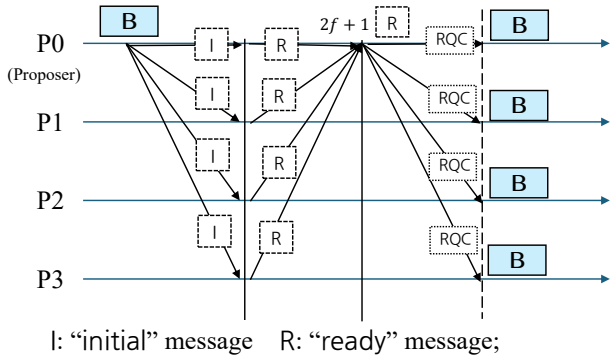


Fig. 7. Consistent broadcast used by the Octopus producer

The Octopus implementation uses the Cobalt ABA [25]. This choice provides a comparatively simple binary primitive and realizes the common coin with threshold signatures. Cobalt is internal to the consumer: the analysis relies only on agreement, binary validity, termination, and decision-evidence soundness.

Octopus identifies each ABA instance by the public session and its exact height or height set and writes its output as (d, e), where e is publicly verifiable decision evidence. Valid evidence binds that identifier and d. No adversary can produce accepted evidence for both bits of one instance except with negligible probability. The evidence authenticates an ABA outcome but is not a second decision value. The consumer treats ABA as a black box and relies only on agreement, binary validity, termination, and this evidence soundness.

APPENDIX C
Octopus PROGRESS PROOF

This appendix proves the construction-specific progress required by Theorem 2. The main text establishes safety from RQC binding, agreement, decision-evidence soundness, and the DPCM compiler's ordered delivery. It remains to show that producer progress eventually triggers consumer coverage and that refinement terminates. We then show that every normal resolution is recovered and every resolved position is eventually delivered.

The lemmas below prove this chain. Correct producers continuously certify queue positions. Certificates in the look-ahead columns eventually raise BATCHREADY even when the current batch contains faulty slots. A terminated root instance either resolves its batch or starts a finite refinement tree. Recovery then turns every normal resolution into an ordered delivery.

Lemma 2 (Correct-Producer Progress). For every correct producer P_i and every queue position (i, k), all correct parties eventually install a valid RQC for (i, k). Consequently, every fixed column eventually contains at least $n - f = 2f + 1$ certified positions at every correct party.

Proof. Proceed by induction on k. Position $(i, 1)$ is initially available, and the correct producer supplies and broadcasts a block. Every correct party eventually stores and validates it

before returning a ready signature, so P_i eventually collects the $n-f=2f+1$ signatures needed for an RQC. It broadcasts the certificate, and certificate relay makes every correct party eventually install it. Installation restores the proposal capacity and enables $(i,k+1)$, for which the producer supplies the next block. The same argument therefore advances through every fixed k. Applying this argument to the position of each correct producer in any fixed column shows that every correct party eventually installs at least $n-f=2f+1$ valid RQCs in that column.

Lemma 3 (Root-Batch Invocation). For $q \geq 0$, let $\mathcal{B}_q = [q\ell n+1,(q+1)\ell n]$. If every correct party eventually delivers through height $q\ell n$, then every correct party eventually raises BATCHREADY for $\mathcal{B}_q$, submits exactly one input to its root ABA_{batch} instance, and obtains a decision with probability one.

Proof. After reaching height $q\ell n$, every correct party identifies $\mathcal{B}_q$ as its next root batch. Its look-ahead window consists of the following ℓ producer columns, as in Algorithm 5. By Lemma 2, each look-ahead column eventually contains certificates for all $n-f=2f+1=\lceil 2n/3\rceil$ correct-producer positions at every correct party. Since the window is finite, condition (ii) of BATCHREADY eventually holds even if faulty producers leave positions in $\mathcal{B}_q$ uncertified. As $\mathcal{B}_q$ uniquely determines the root-instance identifier, all correct parties submit to the same instance. A correct party ignores later triggers after its first submission. Once all correct parties have submitted, ABA_{batch} termination gives the claim.

Lemma 4 (Finite Batch Resolution). Suppose all correct parties invoke the root instance for $\mathcal{B}_q$ and have delivered through height $q\ell n$. Then every height in $\mathcal{B}_q$ is eventually resolved and delivered by every correct party with probability one.

Proof. If the root decides 1, binary validity implies that some correct party input 1 and hence held valid certificates for all positions in $\mathcal{B}_q$. Those certificates were relayed before use. Each certificate has at least $f+1$ correct signers retaining the matching block. A requester that contacts distinct signers therefore reaches a correct holder after at most $f+1$ attempts, and eventual delivery complete every recovery in the finite batch.

If the root decides 0, all correct parties verify the same parent evidence and start the same n row segments. A multi-position segment that decides 0 is split into two deterministic, disjoint, strictly shorter children whose identifiers bind that parent evidence. Every correct party therefore invokes both children without another BATCHREADY condition. ABA termination applies recursively. Each branch has finite depth, and the finite refinement tree ends at segments decided 1 or at singleton decisions. A segment decided 1 has a correct input-1 party and is eventually recovered by the same certificate argument as above. A singleton decided 0 is immediately marked dummy.

Thus every position in the finite batch eventually receives a normal or dummy status. Whenever an outcome is recorded, AdvanceView delivers the longest available consecutive prefix. Because the preceding prefix has already been delivered and every position in $\mathcal{B}_q$ is eventually resolved, each correct party eventually delivers through height $(q+1)\ell n$.

Lemma 5 (Octopus Progress). Under the assumptions of Theorem 2, every fixed height is eventually covered by the consumer and delivered as a normal or dummy entry at every correct party with probability one.

Proof. Induct on the root-batch index q. All correct parties initially have the empty prefix, so Lemma 3 invokes $\mathcal{B}_0$, and Lemma 4 delivers it. If all correct parties have delivered through $q\ell n$, the same two lemmas invoke and deliver $\mathcal{B}_q$. Hence every fixed batch, and therefore every fixed height, is eventually delivered with probability one. The root-invocation lemma supplies participation by all correct parties, while the resolution lemma supplies finite deterministic refinement, recovery, and ordered delivery.

APPENDIX D
DPCM-BACKED APPEND-ONLY BFT STORAGE

This appendix instantiates the DPCM BFT object log as an append-only BFT storage log–$\mathsf{Octopus}^{\mathsf{VID}}$. Unlike general SMR, such a log may order references to retrievable values without replicating every value in full at every party [16]. We replace the CBC producer of Section IV-B with AVID-M, the Merkle-tree-based VID protocol of Yang et al. [28]. DPCM finalizes each predetermined slot as either a certified Merkle root or a dummy entry, while parties retain coded fragments and reconstruct a normal entry's associated value only on demand.

a) AVID-M-based producer: Each slot s identifies one AVID-M instance in which only its owner may disperse. The owner deterministically encodes $\mathbf{B}$ with an $(n-2f,n)$ erasure code and commits to the indexed fragments with a Merkle root r. We rely on AVID-M agreement, availability, correctness, and correct-disperser termination [28].

Standard AVID-M completion does not imply that the completing party locally holds a fragment. We therefore add a holder-backed certification rule. A correct party issues at most one completion share for slot s, and only after completing AVID-M with root r and durably storing its authenticated indexed fragment under r. Any $2f+1$ such shares form a compact certificate π for (s,r). All shares and certificates are domain-separated and bound to the current session. If the owner is correct, at least $n-f=2f+1$ correct parties eventually store their fragments and complete, so the owner obtains enough shares. Correct signers retain their fragments until an authenticated garbage-collection rule permits deletion. This appendix defines no such rule and therefore assumes continued retention.

The DPCM object is r, with the domain-separated commitment $\mathsf{Commit}(r)=H(r)$. Once π is available, Π_P^{store} emits $\textsc{Certified}(s,\mathsf{Commit}(r),\pi)$. The predicate $\mathbf{ValidProd}(s,c,\pi)$ checks $c=\mathsf{Commit}(r)$ and the certificate bound to the session, slot, and root. The event $\textsc{Recover}(s,c,\pi)$ returns r directly. The unchanged consumer

therefore processes only certificates and never reads a coded fragment or payload.

b) Append and on-demand retrieval: To append $\mathbf{B}$, producer i assigns it to its next slot $s = (i, k)$ and invokes the producer above. A normal log outcome exposes (r, π), whereas a dummy entry has no associated payload. Here DPCM recovery returns the ordered object r, whereas storage retrieval subsequently reconstructs the associated block from coded fragments. To retrieve a normal entry, a reader collects $n-2f$ distinct indexed fragments that verify against r, decodes a candidate block, and deterministically re-encodes it. The reader returns the candidate if its encoded Merkle root equals r, and otherwise returns a fixed decoding-error value, as in AVID-M [28].

The re-encoding check is necessary because a Byzantine producer may commit to fragments that do not form a codeword. A block finalized as dummy is not included. Guaranteeing eventual inclusion requires an application-level reproposal mechanism and additional fairness assumptions.

A valid certificate contains shares from at least $f + 1 = n - 2f$ correct fragment holders. Their retention and eventual responses therefore provide enough authenticated fragments for retrieval. The Merkle root binds the indexed fragment vector. If the re-encoding check succeeds, this vector is the deterministic encoding of a unique block, so all correct readers return that block. Otherwise, they return the same fixed error value. For a correct owner, the check succeeds and returns the dispersed block.

c) Scope and comparison: The dispersal and retrieval mechanisms above are those of DispersedLedger [28] and are not DPCM contributions. Let $B = n|\mathbf{B}|$ be the total payload assigned to one producer column. One AVID-M instance sends one coded fragment and Merkle proof to each party, while its all-to-all control messages carry only an $O(\lambda)$-bit root. It therefore costs $O(n^2)$ messages and $O(|\mathbf{B}| + n^2\lambda)$ bits. Completion-share collection and dissemination of the compact certificate remain within these bounds. The n instances in a column cost $O(n^3)$ messages and $O(B + n^3\lambda)$ bits.

BEAT3 [16] uses AVID-FP [22] inside an ACS construction and exposes read/write BFT storage semantics. This appendix provides only an append-only metadata log with lazy reads. The value of DPCM here is the reusable availability–finality boundary and composition proof, not a full key–value storage interface, or an unconditional inclusion guarantee.

Table V compares this construction with BEAT3 under the same metadata-aware accounting. The batched consumer in Octopus^{VID} adds expected $O(n^2/\ell)$ messages and $O(\lambda n^2/\ell)$ bits per column on the optimistic path. If c rows are fully refined, it instead adds expected $O(n^3/\ell + cn^2)$ messages and $O(\lambda n^3/\ell + \lambda cn^2)$ bits, where $c \le f$. BEAT3 runs n ABA instances per ACS epoch, adding expected $O(n^3)$ messages and $O(n^3\lambda)$ bits [16]. Both constructions therefore use expected $O(n^3)$ total messages.

BEAT3 [16] reports $O(B)$ communication by treating the fingerprinted cross-checksum as short metadata. DispersedLedger instead makes its size explicit: AVID-FP has per-party cost $O(|\mathbf{B}|/n+\lambda n^2+\gamma n^2)$ for one dispersal, whereas AVID-M costs $O(|\mathbf{B}|/n + \lambda n)$ [28]. Treating both hash and fingerprint elements as $O(\lambda)$ bits and aggregating over n parties and n proposals gives the table's $O(B + n^4\lambda)$ and $O(B + n^3\lambda)$ availability bounds. Thus both reduce to $O(B)$ in the payload-dominant convention, while the displayed bounds expose their different metadata terms.

TABLE V
EXPECTED METADATA-AWARE NETWORK COST OF Octopus^{VID} AND BEAT3 PER n-PROPOSAL COLUMN OR EPOCH, EXCLUDING LAZY RETRIEVAL. HERE $c \le f$ IS THE NUMBER OF FULLY REFINED ROWS.

	Octopus^{VID} (optimistic)	Octopus^{VID} (pessimistic)	**BEAT3 [16]**
Availability	n AVID-M	n AVID-M	n AVID-FP
Availability bits	$O(n\|\mathbf{B}\| + n^3\lambda)$	$O(n\|\mathbf{B}\| + n^3\lambda)$	$O(n\|\mathbf{B}\| + n^4\lambda)$
Agreement messages	$O(n^2/\ell)$	$O(n^3/\ell + cn^2)$	$O(n^3)$
Total messages	$O(n^3)$	$O(n^3)$	$O(n^3)$
Total bits	$O(n\|\mathbf{B}\| + n^3\lambda)$	$O(n\|\mathbf{B}\| + n^3\lambda)$	$O(n\|\mathbf{B}\| + n^4\lambda)$

The table assumes the compact certificates of Section V-A. Relaying explicit lists of $2f + 1$ signatures would reintroduce an $O(n^4\lambda)$ per-column term. Lazy retrieval, which is excluded from Table V, uses $O(n)$ request/response messages. The $n - 2f = \Theta(n)$ retrieved fragments total $O(|\mathbf{B}|)$ bits; without multiproof aggregation, authenticating them adds $O(n\lambda \log n)$ bits in Merkle paths.

Theorem 3 (DPCM with AVID-M Dispersal). Assume collision-resistant Merkle hashing, a $(2f + 1, n)$-threshold signature secure against forgery, and the AVID-M agreement, availability, termination, and correctness properties [28]. Suppose correct parties follow the completion-share and retention rules above. Then Π_P^{store} satisfies Producer Certified Availability for the metadata object r. Composing it with a consumer satisfying Definition 2 implements an asynchronous append-only BFT storage log. Every normal entry contains one certified root, and all correct readers return the same associated block or the same fixed error value. For a correct slot owner, they return its dispersed block.

Proof. Public verification follows from the threshold certificate bound to the protocol, session, slot, and root. Consider two valid certificates for the same slot. Each required $2f + 1$ shares, so their contributor sets intersect in at least $f + 1$ parties and contain a correct contributor. Because a correct party issues at most one share for the instance, the certificates contain the same root and hence the same commitment. Collision resistance makes H binding for distinct metadata objects. The certificate supplies r directly to RECOVER, so recovery of the DPCM metadata object terminates and returns an object with the certified commitment. Thus public verification, slot binding, and certified availability satisfy Definition 1. Theorem 1 gives integrity, agreement, total order, external validity, data availability, and termination for the metadata log. Finally, every valid certificate contains contributions from at least $f + 1 = n - 2f$ correct fragment holders. Their retention and eventual responses provide enough authenticated fragments for retrieval. Merkle binding and the deterministic re-encoding check give the common payload-level result stated

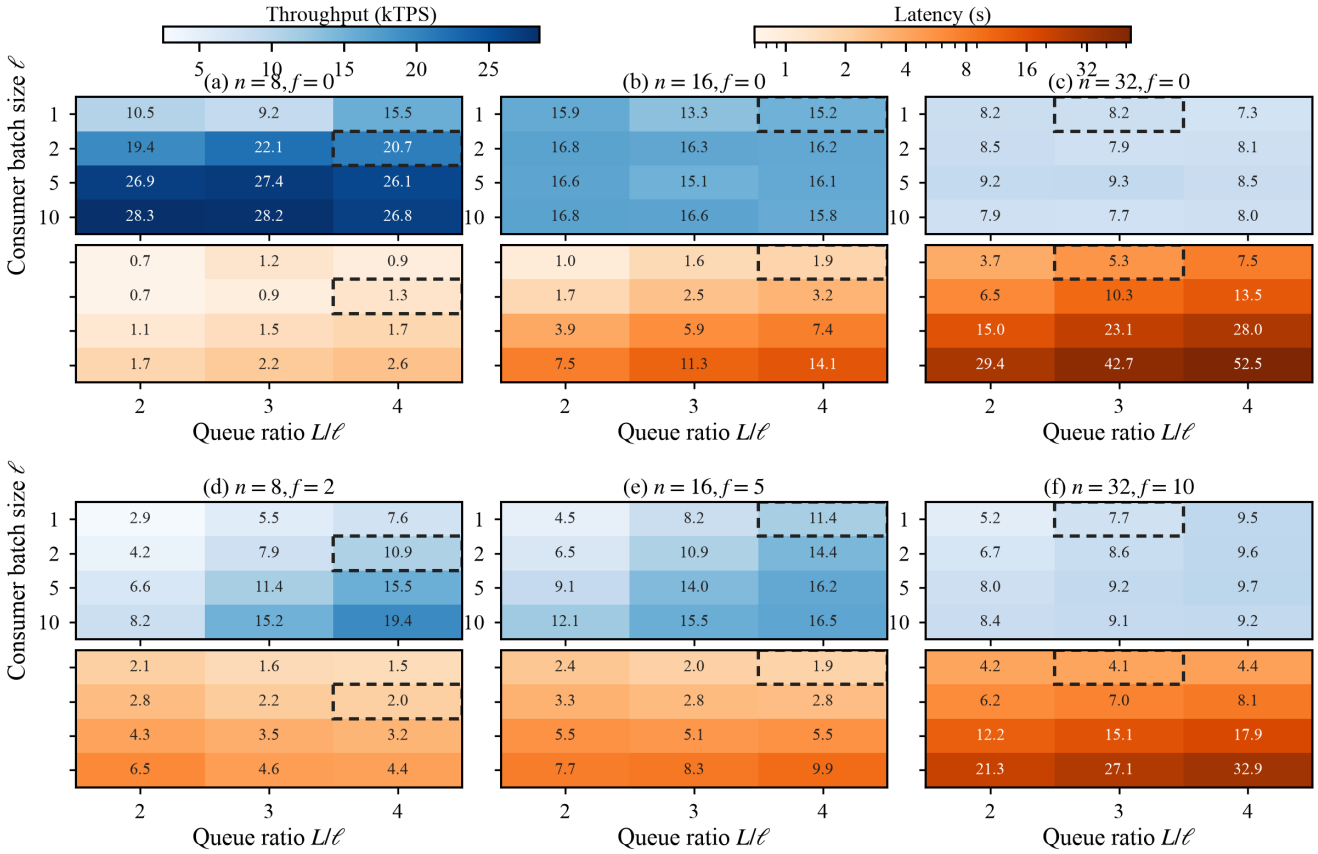


Fig. 8. Throughput and latency of Octopus as the consumer batch size ℓ and queue-length ratio L/ℓ vary. Each panel pairs a throughput heatmap (blue) with a latency heatmap (orange). Columns show $n \in \{8, 16, 32\}$. Panels (a)–(c) are fault-free, and panels (d)–(f) use $f \in \{2, 5, 10\}$ silent faulty parties. Outlined cells denote the parameters used in Figure 6. Each block contains 500 transactions.

above. This additional read property does not place payload recovery inside the consumer.

Appendix E
Parameter Sensitivity

Figure 8 reports the complete 72-setting parameter study described in Section V-B. A larger consumer batch size ℓ allows one agreement instance to process more producer columns, while a larger maximum producer-queue length L permits more CBC instances to run concurrently. Both generally increase throughput but may also increase latency.

At $n = 32, f = 0$, $(\ell, L) = (1, 2)$ gives 3.66 s latency and 88.6% of the highest measured throughput, whereas $(5, 10)$ reaches 99.4% of the highest measured throughput but increases latency to 14.96 s. At $n = 32$, $f = 10$, increasing L from 2 to 4 with $\ell = 1$ raises throughput from 5.2 to 9.5 kTPS while latency changes from 4.17 to 4.39 s.