Proof of Availability & Retrieval in a Modular Blockchain Architecture

Shir Cohen — Technion
Guy Goren — Technion
Lefteris Kokoris-Kogias — IST Austria
Alberto Sonnino — Mysten Labs
Alexander Spiegelman — Aptos

Abstract

This paper explores a modular design architecture aimed at helping blockchains (and other SMR implementation) to scale to a very large number of processes. This comes in contrast to existing monolithic architectures that interleave transaction dissemination, ordering, and execution in a single functionality. To achieve this we first split the monolith to multiple layers which can use existing distributed computing primitives. The exact specifications of the data dissemination part are formally defined by the *Proof of Availability & Retrieval* (PoA&R) abstraction. Solutions to the PoA&R problem contain two related sub-protocols: one that "pushes" information into the network and another that "pulls" this information. Regarding the latter, there is a dearth of research literature which is rectified in this paper. We present a family of pulling sub-protocols and rigorously analyze them. Extensive simulations support the theoretical claims of efficiency and robustness in case of a very large number of players. Finally, actual implementation and deployment on a small number of machines (roughly the size of several industrial systems) demonstrates the viability of the architecture's paradigm.

1 Introduction

Blockchain systems are currently supporting a trillion-dollar economy. New use cases emerge every day and with the promise of "Web 3.0" powering the future digital societies, the number of users grows rapidly. Nevertheless, more than a decade after Bitcoin's invention, blockchains' scalability remains one of the prevalent problems. This problem exists in two dimensions. First, the number of transactions per second a blockchain can process with low latency, enabling real-time payments as well as robustness under high load. Second, the level of decentralization of the system that manages to achieve that high performance. This is important even in permissioned settings since to increase trust blockchains should be as decentralized as possible.

Implementations of blockchain protocols in a permissioned setting are currently using leader-based SMR protocols such as PBFT [15], Tendermint [10], or Hotstuff [39]. Although Tendermint and Hotstuff reduce the total load of the system when the leader is good to $O(n)$, they are still challenging to scale. This is because of the monolithic architecture proposed by current SMR designs, where the leader is expected to propose already executed valid operations and disperse them directly to all parties on the critical path, quickly using up the computing, storage, and networking resources of the leader node.

One good approach to tackle the network bottleneck is to reduce the traffic on the critical path by running consensus on the metadata instead of on the full blocks. This is evident by its abundant use in literature and industry (e.g., [8, 15, 10]). In many works, achieving consensus on the metadata and disseminating the full blocks are deeply intertwined (which may help performance in a particular system but hinders attempts to reuse in other systems). However, these works gain efficiency mostly thanks to not being deployed in real adversarial settings. If, for example, we use a gossip network to disseminate the block like Tendermint [10] or Filecoin [38] then the liveness of the consensus is dependant on the performance and robustness of the gossip network which in their majority are not Byzantine Fault tolerant[1].

Nevertheless, the idea of splitting responsibilities is a natural one. In this paper, we treat it as a first-class citizen generalizing not only to decoupling communication from ordering but also

[1] The single exception is Guerraoui et al. [25] which is yet to be implemented due to its complexity, high runtime, and inability to identify when it fails.

computation. This results in the proposal for a *modular SMR* design. This design, unlike classic SMR, does not directly solve the SMR problem but instead composes existing sub-protocols towards building an SMR. This allows for better usage of resources and exposes a key unexplored bottleneck, that of post-ordering retrieval of data.

More specifically, we split the responsibilities of data dissemination, data ordering, and data execution into different modules. Data dissemination is done through a disperse&retrieve module that can be implemented by any Asynchronous Verifiable Information Dispersal (AVID) protocol [13]. Data ordering is done through any kind of Byzantine Atomic Broadcast (AB) protocol [33, 20, 13] and execution is done through any deterministic execution engine [5, 1, 26].

Once we define this layering approach it becomes apparent that there is a gap of research on the retrieval step. This step is supposed to take the totally ordered proofs of availability that the AB outputs and retrieve the actual data to be executed. Current AVID protocols focus on scaling the disperse phase, but the retrieval protocols either ask the initial source for the data or collect from a supermajority of parties error-corrected chunks. Both of these protocols impose an $O(n)$ cost per process for retrieval and do not try to load-balance. We address this gap in the literature with our scalable retrieval protocol. There we investigate how to efficiently run the retrieve sub-protocol of AVID. Unlike existing designs that cost $O(n)$ messages per node, we show how, using a probabilistic retrieval algorithm, we can achieve complete retrieval with an expected $O(\log n)$ messages per node.

The Proof of Availability & Retrieval Problem

In a nutshell, the PoA&R problem detaches the act of "sending" a block from the part in which nodes "receive" it. Thus, a significant amount of the costs is transferred from the critical path to a time of the recipient's choice. To do so, each block is translated into a (short) proof π and when a node aims to inform the network of a new block of information (or transactions in our blockchain example) it disseminates π instead of the actual block. This can be done, for example, by broadcasting π, which is cheaper than broadcasting the block itself when using an efficient proof generator. A node that receives π stores it and is essentially convinced that when the actual block is needed it will be retrievable.[2]

To obtain the block itself, processes can retrieve it at their own time. In this sub-protocol they reconstruct the initial block, using the stored proof π. Since we alleviate the costs of dispersing the block's evidence into the network, the act of retrieving the block must incur additional costs. However, this kind of paradigm equips systems designers with the flexibility to decide when to undertake such costs. Specifically in blockchains systems, in times of congestion processes can progress by making consensus decisions on proofs alone, whereas the block retrieval and execution can be updated when the load decreases.

In our proposed solution, the creation of the proof π is done using an erasure code scheme and a vector commitment scheme. When a process aims to share a block, it uses erasure coding to create a vector of n code words. It then creates a commitment that binds each word to the entire vector and sends each word (together with the commitment) to a different process. Processes that receive a commitment return a signature to the sender. Once the sender collects "enough" signatures, it forms the proof π that the block can be reconstructed. This is the basic "push" part in several AVID protocols [13, 20, 33].

In existing AVID protocols, retrieving the block (corresponding to the proof π) from the network is done via collecting a large number of code words and reconstructing the block. This might be too costly in large-scale systems. Instead, for the retrieval part, we propose a randomized solution that is deterministically safe and provides liveness with probability 1. Our proposed protocol incorporates vector reconstruction with random sampling. That is, a process that attempts to retrieve a block, occasionally samples a random subset of processes and asks them for the block. Clearly, when processes first try to retrieve the block, the creator of the block is the only process that knows it, thus, more communication rounds are needed. However, the spread of information is typically very fast. This intuitive claim is formally proved in Section 5. Moreover, we analyze different sample sizes that allow for different trade-offs in the cost structure.

Main contributions

- We formalize a modular architecture for the design of blockchains that enables flexibility by using off-the-shelf solutions for parts of the system. In particular, we define the *proof of availability & retrieval* abstraction that is required for our architecture.

[2] Notice that, unlike for AVID, the node does not need to reliably broadcast π. The AB layer takes care of that.

- We recognize a gap in research regarding the retrieval sub-protocol and present a family of (possibly) probabilistic protocols that offer a variety of cost structures. In particular, by using a probabilistic approach, we can reduce the expected cost of messages per node from $\Theta(n)$ to $\Theta(\log n)$.
- We analyze the behavior of our protocols both theoretically and with extensive simulations for large-scale systems. For smaller-sized systems, we implement and deploy our architecture on a network of AWS machines and show its viability in practice.

2 Model

We consider a standard asynchronous message-passing model with Byzantine faults and a computationally bounded adversary [11, 12, 30]. There is a fixed set Π of $n = 3f + 1$ processes, at most f of them are faulty. These faulty processes are called *Byzantine* and are not bound to the protocol. The rest of the processes are *correct* and act according to the protocol. Until a process is corrupted by the adversary it is called *so-far correct.* Each pair of processes is connected via reliable albeit asynchronous links. That is, messages eventually arrive but there is no bound on message delays. We consider an adversary that is exposed to all of the network communication, fully controls the Byzantine processes, and can adaptively choose which processes to corrupt with an after-the-fact removal effect. That is, even after a so-far correct process has sent a message, if the message is yet to be delivered the adversary can view this message, choose to corrupt the sending process, and change/delete the message.

We model the computations made by all system components as probabilistic Turing machines and bound the number of computational basic steps allowed by the adversary by a polynomial in a security parameter λ. We further assume a trusted setup, namely, before the start of the protocol, each party is dealt its own secret key share and the public keys as internal states. This can be achieved by a trusted dealer or distributed key generation protocols [28, 19, 2], but for presentation simplicity, we consider this trusted setup for granted.

Our protocols employ several standard cryptographic primitives with the following abstractions.
Erasure code. We use an erasure code scheme that consists of two algorithms, EC.*encode* and EC.*decode*. EC.*encode* takes a block b and returns a vector of n code words such that any $n - 2f$ out of the n code words suffice for EC.*decode* to fully reconstruct the original block b. (See [9].)
Threshold signature. This scheme allows processes to combine different signatures on the same message, into a single compact signature. It consists of the *SignShare* and $Combine_t$ algorithms. The first is used by each individual to produce its individual signature, while $Combine_t$ is used to produce a single compact signature from t valid individual signatures. Individual/full signatures are $O(\lambda)$-bit long. (See [29].)
Vector Commitment. The vector commitment (VC) scheme is comprised of three algorithms: 1) VectorCommit($\mathbf{c}$) which takes an n-element vector $\mathbf{c}$ and returns a commitment to that vector vc_{sig}; 2) PositionalCommitProof($\mathbf{c}, vc_{sig}, c_i, i$) which takes the vector $\mathbf{c}$, its commitment vc_{sig}, the element c_i and its position i in $\mathbf{c}$, and returns a positional proof π_i; and 3) VerifyElement(vc_{sig}, c_i, π_i) that uses the proof π_i to check whether c_i is indeed the i^{th} element in the vector whose commitment is vc_{sig}. Both vc_{sig} and π_i bit lengths are in $O(\lambda)$. (See [16].)

3 Modular SMR architecture

As the first contribution of this work, we propose a layered architecture for SMR that enables plug-and-play use of PoA&R, Atomic Broadcast, and deterministic execution protocols. We first define the properties of the protocol and then in Figure 1 we provide a breakdown of how it works. In the rest of the paper we focus on the PoA&R protocol. For completeness, we define below the properties the rest of the layers should have.

3.1 The Proof of Availability & Retrieval Problem definition

In this section, we introduce and formally define the Proof of Availability & Retrieval (PoA&R) abstraction. This abstraction should capture the ability to disseminate a block in a fashion that enables reducing the cost on the critical path (the consensus module). Roughly speaking, we detach the act of "sending" a block from the part in which processes "receive" it. Thus, a significant amount of the costs can be transferred from the critical path to a time of the recipient's choice.

The PoA&R abstraction exposes an interface with two operations and two callbacks:

- `PoA_push`(b): an operation invoked by a process to push (disseminate) a proof for block b.
- `PoA_commit`(π): a callback triggered to commit a proof π. (For the availability of a block b.)
- `PoA_pull`(π): an operation invoked by a process to pull (retrieve) a block that corresponds to the proof π.
- `PoA_deliver`(b): a callback triggered to handle the delivery of a block b.

We only define the single-sender problem (with a given known sender p_s), since this specification can easily be extended to the case with multiple senders that push/pull blocks. For the multiple-senders problem, many single-senders instances can be active in parallel (they can be distinguished, for example, by using source tags).

We assume the existence of two functions `CreateProof` and `Verify` that satisfy the following conditions. For any arbitrary blocks b, b_1 and b_2, it holds that `CreateProof`$(b_1) \neq$ `CreateProof`(b_2) iff $b_1 \neq b_2$, and `Verify`$(b, \pi) = \mathit{true}$ if $\pi =$ `CreateProof`(b) and `Verify`$(b, \pi) = \mathit{false}$ otherwise. Given these standard cryptographic functions, the PoA&R problem is defined by the following properties that must be satisfied at all the possible executions.

Definition 1. *Proof of Availability & Retrieval:*

- ***Push-validity:*** *If p_s is correct and invokes* `PoA_push(b)`*, then every correct process eventually performs* `PoA_commit`(π) *such that* `Verify`$(b, \pi) = \text{true}$.
- ***Pull-validity:*** *If a correct process p_i performs* `PoA_deliver(b)`*, then there exists π such that p_i had performed* `PoA_commit`(π) *and* `Verify`$(b, \pi) = \text{true}$.
- ***Pull-termination:*** *Let p_i be a correct process. For each π such that p_i had performed* `PoA_commit`(π) *and has invoked* `PoA_pull`(π)*, p_i eventually delivers a unique block with probability 1.*

Since PoA&R is defined as part of a blockchain architecture, we are able to capture exactly what is necessary without redundant properties. For example, we do not need the agreement property of AVID, which in turn enables us to design more efficient protocols.

Complexity measures A PoA&R protocol satisfies the validity and termination properties even in cases of asynchrony and Byzantine faulty processes, which means it is robust by design. However, executions with failures and asynchrony are not the majority in the routine operation of systems. In fact, the "nice case" in which no failures occur and the network is almost synchronous can be quite common in practice. It is therefore desired to design systems that minimize costs in these common "nice" conditions while allowing for increased costs when having to deal with troubles. We assume that the common-case execution of the considered blockcahin system has the following properties:

- **Good processes.** All process are correct.
- **Synchrony.** The roundtrip of messages in the network is within Δ.
- **Concurrency.** Processes start the pulling sub-protocol at the same time.

The last assumption is crucial for the stochastic analysis of the protocols. It is a justified approximation since in the normal modus operandi a process pulls immediately after the consensus decision, and synchrony causes these decisions to happen within a short time span at almost all processes.

We henceforth use the following (per process) complexity metrics:

Message complexity. The expected number of messages a process sends during a common-case execution.

Bit complexity. The expected number of bits a process sends during a common-case execution.

Round complexity (running time). We define an asynchronous round in the standard way (see [14]). Essentially, this measurement counts the number of messaging "rounds", when the protocol is embedded into a lock-step timing model. The round complexity is then the expected number of asynchronous rounds it takes a process to complete the protocol (deliver a block) during a common-case execution.

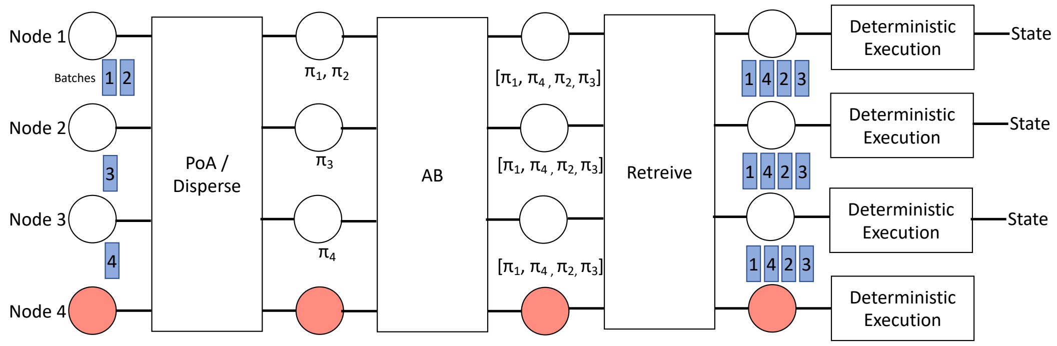


Figure 1: Overview of the Layered SMR approach

3.2 Atomic Broadcast

The classic definition of Atomic Broadcast states that every execution of a protocol solving AB should satisfy:

- **Validity:** If a correct process *broadcasts msg* then all correct processes eventually deliver *msg*.
- **Agreement:** If a correct process *delivers msg* then all correct processes eventually deliver *msg*.
- **Integrity:** *msg* is delivered by a correct process at most once, and only if it was previously broadcast.
- **Total order:** If two correct processes deliver both *msg* and *msg*′, they deliver them in the same order.

However, it is well-known that AB is impossible to solve in an asynchronous model even with one possible crash failure [24]. Since we are dealing with an asynchronous setting with Byzantine failures, these properties must be relaxed. There are varied relaxations and protocols solving them in the literature, e.g., [15, 32, 39, 3]. We leave the choice of desired relaxation and implementing protocol for the system designer, but remark that this crucial choice determines the basic theoretical guarantees provided by the SMR system.

3.3 Execution

The execution layer simply takes as input the total ordered set of operations and updates the state. The only property required by this layer to implement SMR is that of determinism. Solutions such as [35, 18, 17, 23] can be used to provide a scalable execution layer.

3.4 Bringing them all together

Our SMR works in layers. First, every process that has a batch of operations transmits it through PoA&R and collects a proof of availability π. These proofs are then submitted to the AB layer which totally orders the proofs without having to incur the cost of handling the data. The totally ordered proofs are then fed into the Retrieval sub-protocol that recovers any batches not locally available at each process. Once a batch is available and at the head of the ordering queue, the process locally executes it and updates the state. Figure 1 and Algorithm 1 give an overview of the architecture.

4 Proof of Availability and Retrieval Protocols

Many protocols can implement the abstraction of Definition 1, for example, AVID protocols. As we have observed, the retrieval (pulling) part significantly affects the performance of the system. Thus, we propose a PoA&R module with a family of pulling sub-protocols that offer different trade-offs in terms of time vs. communication costs. Clearly, any pulling sub-protocol depends on the dispersal (push) sub-protocol, therefore, it is defined with relation to a given push-commit protocol.

In a trivial PoA&R scheme, when a process wishes to push a block b, it simply sends it to all processes. Upon receiving the block, a correct process commits b as the proof for itself (i.e.,

Algorithm 1: Layered-SMR Calling Sequence

```
Committed_Proofs← {}                          // a set of unordered proofs.
Ordered_Proofs← []                  // proofs that are ordered according to AB.
upon New_Batch(b) do
    PoA_push(b)
upon PoA_commit(π) do
    Committed_Proofs.push(π)
while not Committed_Proofs.empty() do
    π ← Committed_Proofs.pop()
    Atomic_Broadcast(π)
upon AB_deliver(π) do
    Ordered_Proofs.push(π)
    PoA_pull(π)
upon PoA_deliver(b) do
    State ← Execute(b)        // Execute waits for all batches ordered before to
     finish.
```

CreateProof$(b) = b$), and when it wants to pull it, it immediately delivers b. It is straightforward that this simple algorithm satisfies Definition 1 and is also optimal in the number of messages and the round complexity. However, it is far from being optimal in terms of bit complexity. More importantly, this solution does not allow the desired goal of removing the load from the consensus module. That is, processes take the block itself as an input for a single consensus decision. In typical systems, a single block contains a large number of transactions in order to increase throughput, which implies large block sizes. This renders the above sub-protocol impractical for large systems with a high level of decentralization (i.e. large n).

To bypass this problem several works suggested using erasure coding and vector commitments [30, 33] in their protocols, that can be interpreted as push-commit sub-protocols. We use this single (standard) push sub-protocol and focus instead on the pulling sub-protocol. For completeness, we first detail the standard push-commit protocol and briefly explain the standard deterministic pull protocol. We then present a pulling protocol that improves the standard one by satisfying pull-termination with probability 1 instead of deterministically.

4.1 Erasure Coded PoA&R

Push-Commit Protocol. (The pseudo-code appears in Algorithm 2.) In this algorithm, the sender erasure-codes the block b into n code words from which any $n-2f$ words suffice for reconstructing b. These n code words are treated as a vector from hereon. The sender then uses a vector commitments mechanism to create a binding proof for each vector element. Each code word and proof are then sent to a process corresponding to the vector position. A process that receives a valid vector commitment proof, returns to the sender a signed share for a threshold signature on the vector signature (denoted as vc_{sig}). When the sender collects enough shares ($n-f$ this time), it combines them into a valid threshold signature on vc_{sig} and sends that signature to all as the proof. A process that receives a valid threshold signature commits it as a "proof for the availability of a block".

The bit and message complexities of Algorithm 2 are in $\Theta(|B| + n\lambda)$ and $\Theta(n)$ respectively. Since the sender must transmit $\Omega(|B|)$ bits and must send at least f messages to enable the correct reconstruction of b, the Push-Commit protocol is asymptotically optimal in the number of bits it communicates as well as in the number of messages.[3]

Deterministic Pull-Deliver protocol. A natural pull-deliver protocol that complements the presented push algorithm appears in Appendix A. In this algorithm, a process that initiates PoA_pull(π) sends to all other processes a request to reconstruct the block associated with π. Each of the processes answers with its share of the data and the vector commitment proof attached to it. When the puller

[3] We note that the cryptographic primitives for vector commitments are somewhat heavy in local computations and might slow down a system. In comparison, simpler commitment primitives such as Merkle trees [31] can prove a better match as long as n is not "too large". However, they incur a $\Theta(\lambda \log n)$ bit complexity per commitment in comparison to the constant (λ) complexity of the vector commitment primitive.

Algorithm 2: Push-Commit protocol: code for process p_i

```
upon $PoA\_push(b)$ do
    $(c_1, \ldots, c_n) \leftarrow \text{EC}.encode(b)$
    $vc_{sig} \leftarrow \text{VectorCommit}(c_1, \ldots, c_n)$
    for $each\ p_j \in \Pi$ do
        $\pi_j \leftarrow \text{PositionalCommitProof}(b, vc_{sig}, c_j, j)$
        Send$(vc_{sig}, c_j, \pi_j)$ to process $p_j$
upon $Receiving\ (vc_{sig}, c_i, \pi_i)\ from\ p_s\ for\ the\ first\ time$ do
    if $VerifyElement(vc_{sig}, c_i, \pi_i)$ then
        $\sigma_i \leftarrow SignShare(vc_{sig})$
        Send $(ACK, \sigma_i)$ to process $p_s$
upon $Receiving\ (ACK, \sigma_i)\ from\ p_i\ for\ the\ first\ time$ do
    if $VerifyShare(vc_{sig}, i, \sigma_i)$ then
        $\text{Sigs} \leftarrow \text{Sigs} \cup \{\sigma_i\}$
    if $|Sigs| = n - f$ then
        $\sigma \leftarrow Combine_{n-f}(vc_{sig}, \{\sigma_i \in \text{Sigs}\})$    // choose $n - f$ partial signatures and combine them
        Send$(Commit, vc_{sig}, \sigma)$ to all processes
upon $Receiving\ (Commit, vc_{sig}, \sigma)\ from\ p_s\ for\ the\ first\ time$ do
    if $Verify(vc_{sig}, \sigma)$ then
        PoA_commit$(vc_{sig})$
```

collects $f + 1$ valid replays, it reconstructs b. It then verifies that b is valid by computing the vector commitment procedure on b and comparing the resulting vc_{sig} to the one in π. If the block is valid, it can be delivered. Otherwise, deliver $\perp$– indicating that the sender of the block is faulty and no valid block exists.

This algorithm costs $\Theta(|B| + \lambda n)$ bits per puller and is very efficient in moderately large systems where $n < |B|$. For larger-scale systems, however, the linear number of messages per puller might hinder performance. A "strawman" solution to this issue is the following. A puller first asks the sender for the block. If the sender does not respond timely, then the puller initiates the deterministic pull-deliver protocol. Although this protocol seems to cost on average only a single message and $O(|B|)$ bits per puller, it fails in practice because many pullers ask the sender for the block concurrently, thus causing it to stall and become a fatal bottleneck. This is because there is a process (the sender) that experiences an $\Omega(n)$ message and $\Omega(n|B|)$ bits complexity. The acute imbalance of costs leads to a severe bottleneck in large systems. We deal with this imbalance problem by proposing a family of randomized pull-deliver protocols. These protocols combine rumor spreading in a "reverse gossiping" manner for common-case performance together with erasure-code reconstruction to ensure safety.

Probabilistic Pull-Deliver protocols. (The pseudo-code is divided between Algorithms 3 and 4.) A process that initiates `PoA_pull`(π) and does not have the block locally does the following. It flips a (biased) coin with a probability of k/n of getting heads. If heads is flipped, then the puller sends a reconstruction request to all. Regardless of the coin's outcome, the puller randomly selects a set $\mathcal{S}$ of k processes and sends them a block request (for the transmission of the block associated with π). A process that receives a reconstruction request answers with its code word. A process that receives a "block-transmission" request answers with the block if it has it, otherwise, it informs the puller that it does not have the block (via a "NACK" message). If the puller receives a "NACK" from a process $p_j \in \mathcal{S}$, it removes p_j from $\mathcal{S}$ and randomly chooses a new process, sends this process a block-transmission request, and adds it to $\mathcal{S}$. If the puller does not receive any reply from $p_j \in \mathcal{S}$ within some predefined time (say Δ), it randomly chooses a process not in $\mathcal{S}$, sends this process a block-transmission request, and adds it to $\mathcal{S}$. After every new k block requests, the puller flips the coin again to decide whether to attempt a reconstruction from all or not.

Algorithms 3 and 4 offer a variety of cost structures for the system designer to choose from. The cost is comprised of the expected message, bit and round complexities in the common case. These

Algorithm 3: Probabilistic pulling protocol - part 1: code for puller process (p_{pull}).

```
upon PoA_pull($vc_{sig}$) do
    if there exists a block $b$ in memory such that Verify($vc_{sig}, b$) = True then
        PoA_deliver($b$)                                   // from local memory
    $InTransit \leftarrow \{\}$
    $NewInTransit \leftarrow \{\}$
    $counter \leftarrow 0$
    while did not PoA_delivered a block corresponding to $vc_{sig}$ do
        goto SendReq procedure
SendReq procedure :
    if $|InTransit| < f + k$ then
        for each $\langle p_j, time \rangle \in NewInTransit$ do
            if $currTime > time + \Delta$ then
                $NewInTransit \leftarrow NewInTransit \setminus \{\langle p_j, time \rangle\}$
        $\hat{k} \leftarrow k - |NewInTransit|$
        randomly choose a set $\mathcal{S} \subseteq \Pi \setminus InTransit$ of $\hat{k}$ processes
        for each $p_j \in \mathcal{S}$ do
            if $counter \ \% \ k = 0$ then
                with probability $\frac{k}{n}$ Send($RECONSTRUCT, vc_{sig}, \sigma$) to all
            $counter = counter + 1$
            $NewInTransit \leftarrow NewInTransit \cup \{\langle p_j, currTime \rangle\}$ // currTime is the reading
             of the puller's local clock at the point of sending the message.
        $InTransit \leftarrow InTransit \cup \mathcal{S}$
        Send (PULL, $vc_{sig}, \sigma$) to $\mathcal{S}$
upon Receiving ($ACK, b$) from process $p_j$ do
    if Verify($vc_{sig}, b$) then
        PoA_deliver($b$)
upon Receiving ($NACK$) from process $p_j$ do
    $InTransit \leftarrow InTransit \setminus \{p_j\}$
    $NewInTransit \leftarrow NewInTransit \setminus \{\langle p_j, \cdot \rangle\}$
upon Receiving ($vc_{sig}, c_j, \pi_j$) from $p_j$ for the first time do
    if VerifyVecotr($vc_{sig}, i, c_j, \pi_j$) = True then
        CODEDVECTOR[$j$] $\leftarrow c_j$
        if $|CODEDVECTOR| > f$ then
            $b \leftarrow$ EC.decode(CODEDVECTOR)
            if Verify($vc_{sig}, b$) then
                PoA_deliver($b$)
            else
                PoA_deliver($\perp$)
```

complexities are determined by the choice of k, as we show in the theoretical analysis in Section 5. While using our probabilistic pulling protocols cannot significantly reduce the bit complexity in comparison to the deterministic counterpart, in terms of expected message complexity we can gain an exponential improvement. Specifically, we prove that for $k \in \Theta(1)$ we get a message complexity in $O(\log n)$, for $k \in \Theta(\log n)$ the message complexity is in $O\left(\frac{\log^2 n}{\log \log n}\right)$, and in for $k \in \Theta(\sqrt{n})$ we get a message complexity in $O(\sqrt{n})$. However, the reduced message complexity does not come for free. Either the round complexity increases (for $k \in \Theta(1)$), or the bit complexity increases (for $k \in \Theta(\sqrt{n})$). Therefore, different choices of k fit different systems according to where the system bottleneck is.

Algorithm 4: Probabilistic pulling protocol - part 2: code for all processes.

```
1 upon Receiving $(PULL, vc_{sig}, \sigma)$ from $p_{pull}$ do
2     if previously performed PoA_deliver(b) for the commitment $vc_{sig}$ then
3         Send $(ACK, b)$ to $p_{pull}$                          // $b$ can be $\bot$
4     else
5         Send $NACK$ to $p_{pull}$
6 upon Receiving $(RECONSTRUCT, vc_{sig}, \sigma)$ from $p_{pull}$ do
7     if $Verify(vc_{sig}, \sigma)$ then
8         if previously received a valid $(vc_{sig}, c_i, \pi_i)$ then
9             Send$(vc_{sig}, c_i, \pi_i)$ to $p_{pull}$
```

4.2 Correctness Proof

The proof relates to the combination of algorithm 2 for pushing a proof with algorithms 3 and 4 for pulling the block.

Push-validity: If p_s is correct and invokes `PoA_push`(b), then every correct process eventually performs `PoA_commit`(π) such that `Verify`$(b, \pi) =$ *true.*
A correct sender that initiates `PoA_push`(b), sends correct code words, positional commitments and vector commitment to all on line 6. Every correct process eventually receives the sender's message, signs vc_{sig} and sends it to p_s. Process p_s eventually receives at least $n - f > 2f$ such correct partial signatures, hence, it is able to combine them into a threshold signature on line 14. As the sender is correct, it sends a correctly structured tuple $(Commit, vc_{sig}, \sigma)$ to all. Consequently, every correct process receives $(Commit, vc_{sig}, \sigma)$ from p_s exactly once and therefore, eventually performs `PoA_commit`(vc_{sig}, σ) on line 19. The vector commitment mechanism guarantees that `Verify`$(b, vc_{sig}) =$ *true.*

Pull-validity: If a correct process p_i performs `PoA_deliver`(b), then there exists π such that p_i had performed `PoA_commit`(π) and `Verify`$(b, \pi) =$ *true.*
Let p_i be a process according to the above. It performs `PoA_deliver`$(b \neq \bot)$ on line 14 or on line 24. This only happens if `Verify`$(b, \pi \triangleq vc_{sig}) =$ *true* where vc_{sig} is known to p_i because it was previously committed.

Pull-termination: Let p_i be a correct process. For each π such that p_i had performed `PoA_commit`(π) and has invoked `PoA_pull`(π), p_i eventually delivers a unique block with probability 1.
Let p_i be a process according to the above. Process p_i is correct and commits only after receiving a valid threshold signature on vc_{sig} (lines 18-19). This implies that $n - f$ processes have signed vc_{sig} on line-9, out of these $n - f$ at least $f + 1$ are correct processes that have correct code words from the pusher. If p_i receives the code words of these $f + 1$ correct processes it succeeds in reconstructing the block (based on the erasure coding scheme) on line 22. Therefore, once p_i performs a RECONSTRUCT broadcast, it will eventually receive enough correct code words to reconstruct and deliver the block on line 24. As long as p_i haven't PoA_delivered the associated block, it performs this broadcast with constant probability every Δ (or even more frequently if it receives NACKs). Therefore, the probability of eventually performing the broadcast (and eventually PoA_delivering a block) is 1.

5 Theoretical Analysis

We analyze the complexity of the common-case in which all processes attempt to synchronize at the same time, the sender is correct, and in addition, the network is in a stable "nice" period. Concretely, we analyze the complexity in cases where no faults occur and a message round-trip time takes exactly 1 time-unit throughout the network.

5.1 One Sample per Round

With $k = 1$ (a single sample per round), our model resembles the random phone-call model of [21]. There is an elegant analysis for address-oblivious rumor spreading in this model that was made by Karp, Schindelhauer, Shenker, and Vocking in [27]. Our analysis is inspired by their techniques and therefore shares similar structure. Nevertheless, their analysis yields slightly different quantities than ours, since they consider a protocol in which processes both actively tell the rumor (send the block) as well as passively inform others who ask for the rumor. In contrast, we allow only to passively inform those who ask. Moreover, the analysis in [27] only holds for large enough n, a restriction we do not have since we bound the expected values rather than the probability of higher costs.

Theorem 1. *In a common-case execution of Algorithms 3 and 4 with $k = 1$, the pulling terminates within $O(\log n)$ expected rounds.*

Proof. The spread of information can be modeled as a Markov process, with states $\{1, \ldots, n\}$ which represent how many process currently have the block. Denote the random variable $X_r \in \{1, \ldots, n\}$ to be the number of informed processes after round r. Given X_r we have that $\Delta_{r+1} \triangleq X_{r+1} - X_r$ follows a binomial distribution with $n - X_r$ experiments and a success probability of $\frac{X_r}{n-1}$ per experiment. I.e, $\Delta_{r+1} \mid X_r \sim B(n - X_r, \frac{X_r}{n-1})$, and

$$\mathbb{E}[X_{r+1} \mid X_r] = X_r + \mathbb{E}[\Delta_{r+1} \mid X_r] = X_r + \frac{n}{n-1}X_r - \frac{1}{n-1}X_r^2 = X_r\left(2 + \frac{1}{n-1} - \frac{X_r}{n-1}\right) > X_r\left(2 - \frac{X_r}{n-1}\right). \tag{1}$$

For $X_r \leq \frac{1}{2}(n-1)$, we get

$$\mathbb{E}[X_{r+1} \mid X_r] \geq X_r \cdot 1.5, \tag{2}$$

and by the law of total expectation,

$$\mathbb{E}[X_{r+1}] \geq 1.5\mathbb{E}[X_r]. \tag{3}$$

Applying the same argument recursively, yields,

$$\mathbb{E}[X_{r+1}] \geq (1.5)^{r+1}\mathbb{E}[X_0]. \tag{4}$$

Let r_{half} be the first round in which $X_r > \frac{1}{2}(n-1)$. Then by (4) $\mathbb{E}[r_{half}] \leq \frac{\log n}{\log 1.5}$. Now, for $r \geq r_{half}$ denote by Y_r the random variable $n - X_r$. We have that $Y_{r+1} \mid Y_r \sim B\left(Y_r, \frac{Y_r - 1}{n-1}\right)$, and

$$\begin{aligned}
\mathbb{E}[Y_{r+1} \mid Y_r] &= \frac{Y_r^2 - Y_r}{n-1}, \\
\mathbb{E}[Y_{r+1}^2 \mid Y_r] &= \frac{Y_r(Y_r - 1)}{n-1}\left(1 - \frac{Y_r - 1}{n-1}\right) + \left(\frac{Y_r(Y_r-1)}{n-1}\right)^2 = \frac{Y_r^2 - Y_r}{n-1} - \frac{Y_r^2 - Y_r}{n-1}\frac{Y_r - 1}{n-1} + \left(\frac{Y_r^2 - Y_r}{n-1}\right)^2 \\
&= \mathbb{E}[Y_{r+1} \mid Y_r] - \frac{Y_r - 1}{n-1}\mathbb{E}[Y_{r+1} \mid Y_r] + \mathbb{E}[Y_{r+1} \mid Y_r]^2.
\end{aligned} \tag{5}$$

Using the law of total expectation and both of the above equations, we get

$$\begin{aligned}
\mathbb{E}[Y_{r+1} \mid Y_r] &= \frac{\mathbb{E}[Y_r \mid Y_{r-1}] - \frac{Y_{r-1}-1}{n-1}\mathbb{E}[Y_r \mid Y_{r-1}] + \mathbb{E}[Y_r \mid Y_{r-1}]^2 - \mathbb{E}[Y_r \mid Y_{r-1}]}{n-1} \\
&= \frac{\mathbb{E}[Y_r \mid Y_{r-1}]^2 - \frac{Y_{r-1}-1}{n-1}\mathbb{E}[Y_r \mid Y_{r-1}]}{n-1} \leq \frac{\mathbb{E}[Y_r \mid Y_{r-1}]^2}{n-1}.
\end{aligned} \tag{6}$$

Applying (6) recursively we obtain

$$\mathbb{E}\left[\frac{Y_{r+1} \mid Y_r}{n-1}\right] \leq \left(\frac{\mathbb{E}[Y_{r+1-i} \mid Y_{r-i}]}{n-1}\right)^{2^i}. \tag{7}$$

Recall that $Y_r = n - X_r$ and that for $r \geq r_{half}$ it holds that $Y_r < \frac{n+1}{2} \leq \frac{n}{2}$ (since Y_r is an integer). Thus, we can use (5) and get

$$\mathbb{E}[Y_{r_{half}+1} \mid Y_{r_{half}}] = \frac{Y_{r_{half}}}{n-1}(Y_{r_{half}} - 1) \leq \frac{n/2}{n-1}\frac{n-2}{2} \leq \frac{n}{2n}\frac{n-1}{2} = \frac{n-1}{4}. \tag{8}$$

Plugging (8) into (7), we obtain

$$\mathbb{E}\left[\frac{Y_{r+1} \mid Y_r}{n-1}\right] \leq \left(\frac{\mathbb{E}[Y_{r_{half}+1} \mid Y_{r_{half}}]}{n-1}\right)^{2^{(r-r_{half})}} \leq \left(\frac{1}{4}\right)^{2^{(r-r_{half})}}. \tag{9}$$

The above means that by the law of total expectation

$$\mathbb{E}[Y_r \mid Y_{r_{half}}] \leq (n-1)\left(\frac{1}{2}\right)^{2^{(r-r_{half})}}, \tag{10}$$

and the expected additional number of rounds to reach $Y_r \leq 1$ once round r_{half} was reached is $O(\log\log n)$.

Finally, denote by r_{end} the round at the end of which all processes have been informed. We recall that if $Y_r \leq 1$ then $Y_{r+1} = 0$ deterministically. As a result, the linearity of expectation yields

$$\begin{aligned}\mathbb{E}[r_{end}] &\leq 1 + \mathbb{E}[r_{end} - r_{half}] + \mathbb{E}[r_{half}] \\ &= 1 + O(\log\log n) + O(\log n),\end{aligned} \tag{11}$$

and $\mathbb{E}[r_{end}] \in O(\log n)$. □

From Theorem 1 we immediately get the following.

Corollary 1. *In the common-case,*

1. *the expected number of messages per process is in $O(\log n)$, and*
2. *the expected number of bits per process is in $O(|B| + \lambda \log n)$ with only the sender having a higher load of $\Theta(|B|\log n + \lambda \log n)$.*

We remark that since we use only passive spreading without actively gossiping, our expected bit complexity is better than that of [27] which is $\Theta(|B|\log\log n + \log n)$ per receiving process and $\Theta(|B|\log n + \log n)$ for the sender. Moreover, we are able to bypass the lower bound for address-oblivious protocols which is also presented in [27]. We do so by analysing the expected cost rather than the cost w.h.p. Applying a Chernoff bound on our result will show that we are optimal for the cost w.h.p.

5.2 Sampling $\log n$ per Round

For a different trade-off, one may choose the pulling protocol with $k \in \Theta(\log n)$. We show here the resulting expected costs of such choice.

Theorem 2. *In a common-case execution of Algorithms 3 and 4 with $k = \log n$, the pulling terminates within $O\left(\frac{\log n}{\log\log n}\right)$ expected rounds.*

Proof. The spread of information can be modeled by a Markov process, with states $\{1, \dots, n\}$ which represent how many process currently have the block. Denote the random variable $X_r \in \{1, \dots, n\}$ to be the number of informed processes at the end of round r and $Y_r \triangleq n - X_r$ is the number of uninformed processes at the end of round r. Observe that $X_r \geq X_{r-1}$, $X_0 = 1$, and that if $Y_r = 1$ then $Y_{r+1} = 0$ deterministically. Given X_r we have that $\Delta_{r+1} \triangleq X_{r+1} - X_r$ follows a binomial distribution with $n - X_r$ experiments and some success probability P_r. I.e., $\Delta_{r+1} \mid X_r \sim B(n - X_r, P_r)$, and we wish to bound P_r from below.

For each of the $Y_r = n - X_r$ experiments we denote by $\mathcal{S}$ the sampled set of processes. $|\mathcal{S}| = \log n$ and the samples are without replacement which increases the hitting probability. Therefore, P_r is bounded from below by sampling with replacement.

$$\begin{aligned}P_r &= P(\text{at least one out of } \log n \text{ samples without replacement hits one of } X_r \text{ options}) \\ &\geq P(\text{at least one out of } \log n \text{ samples with replacement hits one of } X_r \text{ options}) \triangleq \tilde{P}_r.\end{aligned} \tag{12}$$

By the inclusion–exclusion principle

$$\begin{aligned}\tilde{P}_r =& P\Big(\bigcup_{i=1}^{\log n} \text{a sample from } n-1 \text{ possibilities hits one of } X_r \text{ options}\Big) \\ &- P(\text{at least two samples from } n-1 \text{ possibilities hits one of } X_r \text{ options}) \\ &\geq \log n \cdot \frac{X_r}{n-1} - \binom{\log n}{2}\left(\frac{X_r}{n-1}\right)^2 = \frac{X_r}{n-1}\left(\log n - \frac{(\log n)(\log n - 1)}{2} \cdot \frac{X_r}{n-1}\right),\end{aligned} \tag{13}$$

where the last inequality is due to the union bound which implies that the probability of at least two samples hitting is at most $P\left(\bigcup_{i=1}^{\binom{\log n}{2}} \left(\frac{X_r}{n-1}\right)^2\right)$. Now, for $X_r \leq \frac{n}{\log n}$ we have that

$$\tilde{P}_r \geq \log n \cdot \frac{X_r}{n-1}\left(1 - \frac{\log n - 1}{2} \cdot \frac{X_r}{n-1}\right) \geq \log n \cdot \frac{X_r}{n-1}\left(1 - \frac{1}{2}\right) = \frac{\log n}{2} \cdot \frac{X_r}{n-1}. \tag{14}$$

And using the expectation of a binomial variable, we obtain

$$\begin{aligned} \mathbb{E}[X_{r+1} \mid X_r] = X_r + \mathbb{E}[\Delta_{r+1} \mid x_r] = X_r + (n - X_r)P_r = (1 - P_r)X_r + n \cdot P_r \geq n \cdot P_r \geq n \cdot \tilde{P}_r \\ \geq n \cdot \frac{\log n}{2} \cdot \frac{X_r}{n-1} \geq \frac{\log n}{2} \cdot X_r, \end{aligned} \tag{15}$$

and by the law of total expectation

$$\mathbb{E}[X_{r+1}] \geq \frac{\log n}{2} \cdot \mathbb{E}[X_r]. \tag{16}$$

Let r_1 be the first round at the end of which $X_r \geq \frac{n}{logn}$. By applying (16) recursively we have

$$n \geq \mathbb{E}[X_{r_1}] \geq \left(\frac{\log n}{2}\right)^{r_1} \cdot \mathbb{E}[X_0]. \tag{17}$$

Taking the log of both sides yields

$$\begin{aligned} \log n &\geq r_1 \cdot \log\left(\frac{\log n}{2}\right) \\ r_1 &\leq \frac{\log n}{\log\log n - 1}. \end{aligned} \tag{18}$$

We thus have that $\mathbb{E}[r_1] \in O\left(\frac{\log n}{\log\log n}\right)$.

We now turn to analyze the behavior of $Y_r \triangleq n - X_r$. It follows a binomial distribution $Y_{r+1} \mid Y_r \sim B\left(Y_r, Q_r\right)$, where Q_r is the probability that all of the $\log n$ samples miss. Again we bound it using sampling with replacement and get

$$Q_r \leq \left(\frac{Y_r - 1}{n-1}\right)^{\log n} \leq \left(\frac{Y_r}{n}\right)^{\log n}. \tag{19}$$

Recall that at the end of round r_1 it holds that $X_{r_1} \geq \frac{n}{\log n}$ and therefore,

$$Q_{r_1} \leq \left(\frac{Y_{r_1}}{n}\right)^{\log n} \leq \left(\frac{n - n/\log n}{n}\right)^{\log n} = \left(1 - \frac{1}{\log n}\right)^{\log n} \leq \frac{1}{e}. \tag{20}$$

This, in turn, implies

$$\mathbb{E}[Y_{r_1+1} \mid Y_{r_1}] = Y_{r_1} \cdot Q_{r_1} \leq \left(n - \frac{n}{\log n}\right) \cdot \frac{1}{e} \leq \frac{n}{2}. \tag{21}$$

We denote the first round at which $Y_r \leq \frac{n}{2}$ by r_2. According to the above, it is expected that $r_2 - r_1 \in O(1)$.

Moreover, denote the round when $Y_r \leq 1$ by r_3. We have that

$$\mathbb{E}[Y_{r_2+1} \mid Y_{r_2}] = Y_{r_2} \cdot Q_{r_2} \leq Y_{r_2}\left(\frac{Y_{r_2}}{n}\right)^{\log n} \leq \frac{n}{2} \cdot \left(\frac{1}{2}\right)^{\log n} = \frac{n}{2} \cdot \frac{1}{n} \leq 1. \tag{22}$$

Clearly, $\mathbb{E}[r_3 - r_2] \in O(1)$. Finally, denote by r_{end} the round at the end of which all processes have been informed. We recall that if $Y_r \leq 1$ then $Y_{r+1} = 0$ deterministically. As a result, the linearity of expectation yields

$$\begin{aligned} \mathbb{E}[r_{end}] &\leq 1 + \mathbb{E}[r_3] = 1 + \mathbb{E}[r_3 - r_2] + \mathbb{E}[r_2 - r_1] + \mathbb{E}[r_1] \\ &= 1 + O(1) + O(1) + O\left(\frac{\log n}{\log\log n}\right), \end{aligned} \tag{23}$$

and $\mathbb{E}[r_{end}] \in O\left(\frac{\log n}{\log\log n}\right)$. □

This result implies:

Corollary 2. *In the common-case,*

1. *the expected number of messages per process is in* $O\left(\frac{\log^2 n}{\log\log n}\right)$, *and*
2. *the expected number of bits per process is in* $O\left(|B|\log n + \frac{\lambda \log^2 n}{\log\log n}\right)$.

5.3 Sampling $\sqrt{n}$ per Round

For the fastest termination, that is within $O(1)$ expected asynchronous rounds, it is possible to use our retrieval protocol with $k \in \Theta(\sqrt{n})$ samples. To prove this we use a Markov process, similarly to the previous proofs, with a binomial state-transfer distribution. Specifically, $\Delta_{r+1} \mid X_r \sim B(n - X_r, P_r)$ where we bound P_r to be at least $1 - e^{-\frac{X_r}{\sqrt{n}}}$. Roughly speaking, since $\mathbb{E}[X_r] \in \Omega(\sqrt{n})$, we will get that, in expectation, all processes complete their pull in a constant number of rounds.

Theorem 3. *In a common-case execution of Algorithms 3 and 4 with* $k = \sqrt{n}$, *the pulling terminates within* $O(1)$ *expected rounds.*

Proof. We use the same notation as before, that is, X_r is the number of informed processes at the end of round r and $Y_r \triangleq n - X_r$ is the number of uninformed processes at the end round r. Remember that $X_{r+1} \geq X_r$, $X_0 = 1$, and that if $Y_r = 1$ then $Y_{r+1} = 0$ deterministically. We also denote by $\mathcal{X}_r$ and $\mathcal{Y}_r$ the sets informed and uninformed processes at the end of round r. Again, we have that $\Delta_{r+1} \mid X_r \sim B(n - X_r, P_r)$, and we wish to bound P_r from below. For each of the Y_r experiments we denote by $\mathcal{S}$ the sampled set of processes. We then have that $P_r \geq 1 - P(\mathcal{S} \cap \mathcal{X}_r = \emptyset \mid \mathcal{X}_r)$, and by a simple counting argument we get

$$P(\mathcal{S} \cap \mathcal{X}_r = \emptyset \mid \mathcal{X}_r) = \frac{\binom{n-X_r-1}{\sqrt{n}}}{\binom{n-1}{\sqrt{n}}}. \tag{24}$$

Denote by r_1 the first round at the end of which $X_r \geq \sqrt{n}$. Clearly, $P_1 \geq \frac{\sqrt{n}}{n-1}$ and $\mathbb{E}[X_1] > \sqrt{n}$ hence, $\mathbb{E}[r_1] \in O(1)$.

We further analyze Eq. (24) to get a lower bound on P_r

$$\begin{aligned} P(\mathcal{S} \cap Xr = \emptyset \mid \mathcal{X}_r) &= \frac{(n - X_r - 1)!}{(n - X_r - \sqrt{n} - 1)!\sqrt{n}!} \cdot \frac{(n - \sqrt{n} - 1)!\sqrt{n}!}{(n-1)!} = \frac{(n - X_r - 1)!}{(n - X_r - \sqrt{n} - 1)!} \cdot \frac{(n - \sqrt{n} - 1)!}{(n-1)!} \\ &= \prod_{i=1}^{\sqrt{n}} \frac{n - X_r - i}{n - i} \leq \left(\frac{n - X_r}{n}\right)^{\sqrt{n}} = \left(1 - \frac{X_r/\sqrt{n}}{\sqrt{n}}\right)^{\sqrt{n}} \leq e^{-\frac{X_r}{\sqrt{n}}}. \end{aligned} \tag{25}$$

According to the above, when $X_r \geq \sqrt{n}$ it holds that $P_r \geq 1 - e^{-1}$. Therefore, given that we have reached r_1, we have

$$\mathbb{E}[X_{r_1+1} \mid X_{r_1}] \geq X_{r_1} + (n - X_{r_1})(1 - e^{-1}) > n(1 - e^{-1}). \tag{26}$$

Denote by r_2 the first round at the end of which $X_r > n(1 - e^{-1})$. By Eq. (26) we have that $\mathbb{E}[r_2 - r_1] \in O(1)$.

Recall that $Y_{r+1}|X_r \sim B(Y_r, 1 - P_r)$. Therefore, Eq. (25) can be used to show

$$\mathbb{E}[Y_{r+1} \mid Y_r] \leq Y_r \cdot e^{-\frac{X_r}{\sqrt{n}}}, \tag{27}$$

and for $X_r > n(1 - e^{-1})$ it holds that

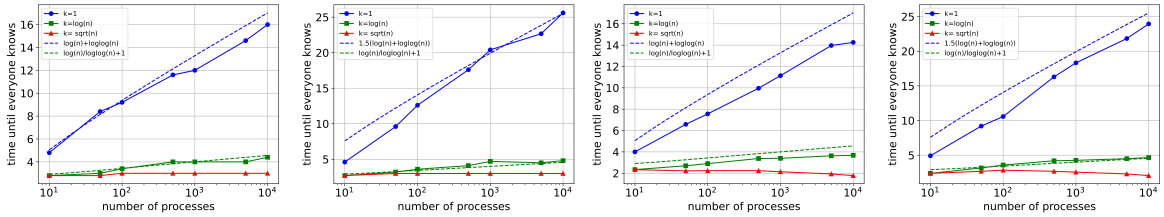
$$\mathbb{E}[Y_{r+1} \mid Y_r] \leq \frac{n/e}{e^{(1-e^{-1})\sqrt{n}}} \leq 1. \tag{28}$$

Now, denote by r_3 the first round at the end of which at most a single process is uninformed, i.e., $Y_{r_3} \leq 1$. By Eq. (28) we have that $\mathbb{E}[r_3 - r_2] \in O(1)$.

Finally, denote by r_{end} the round at the end of which all processes have been informed. We recall that if $Y_r \leq 1$ then $Y_{r+1} = 0$ deterministically. As a result, the linearity of expectation yields

$$\begin{aligned} \mathbb{E}[r_{end}] &\leq 1 + \mathbb{E}[r_3] = 1 + \mathbb{E}[r_3 - r_2] + \mathbb{E}[r_2 - r_1] + \mathbb{E}[r_1] \\ &= 1 + O(1) + O(1) + O(1), \end{aligned} \tag{29}$$

and $\mathbb{E}[r_{end}] \in O(1)$. □



(a) Synchronous common-case.

(b) Synchronous with faults.

(c) Asynchronous no failures.

(d) Asynchronous with faults.

Figure 2: Simulation results for the retrieval sub-protocol in different systems and under different network assumptions. The x-axis states the number of processes n, and the y-axis states the time in units of Δ (the expected roundtrip delay). The graphs depict the time at which the last correct process had delivered the block as a function of n. The network assumption are: (a) The assumed common case, i.e., synchrony and no failures; (b) Synchrony but 1/3 non-responsive processes; (c) Asynchronous delay that follows a Poisson distribution with parameter Δ and no failures; and (d) (c) Asynchronous delay that follows a Poisson distribution with parameter Δ with a 1/3 of the processes that are non-responsive. In all cases, the system sizes vary between 10 to 10^4 processes.

The consequent message and bit complexities for a process are as follows.

Corollary 3. *In the common-case,*

1. *the expected number of messages per process is in $O(\sqrt{n})$, and*
2. *the expected number of bits per process is in $O(|B|\sqrt{n})$.*

5.4 Simulations

We complement the rigorously proven complexities with extensive simulations for systems with a large number of participants. All of the simulations begin with only a randomly chosen sender that posses the block while all other processes have their corresponding code word. We measure the time at which the last process is informed (i.e., delivers the block). For each system, we run 5 simulations and average the end results. The outcome is on par with the theoretical expectations which are depicted by the dashed lines. Moreover, since our protocols are address oblivious and do not rely on synchrony for correctness, they are very robust by design. To demonstrate this, we have also simulated a degraded form of asynchrony by employing stochastic delays that follow a Poisson distribution and set Δ to be the expected delay. Besides the fact that it allows for unbounded delays, the choice of the distribution is arbitrary. (We make no claim as to what best models delays in practical networks.) The results in Figures 2c and 2d suggest that, for Δ that equals the expected delay, the protocols are robust to asynchrony and achieve essentially the same complexities as in synchronous settings. There is even a slight improvement in comparison to synchronous networks, possibly because fast processes are able to answer slower processes in the same "asynchronous round" when they first obtain the block. Finally, we have also simulated the protocol's behavior under faults. Specifically, we run simulations in which a random 1/3 of the processes have crashed. The results appear in Figures 2b and 2d. Again, the simulations indicate the robustness of our protocols, with only a $\frac{3}{2}$x slowdown in performance which is expected since on average third of the samples are wasted on faulty processes. To conclude, our simulations suggest that the pulling sub-protocol is as efficient as expected and is robust under different network conditions.

6 Implementation and Evaluation

We demonstrate the benefits of our approach by providing an implementation, called Layered-SMR. We then evaluate its performance in realistic conditions to demonstrate its real-world value. Many practical systems typically run with small number of nodes, ranging from 10 to 30 [22, 34, 37]. This section demonstrates that despite our retrieval protocol targets very large systems (see Section 5.4), it also provides significant benefits for current real-world deployments.

6.1 Implementation

We implement Layered-SMR on top of a high-performance open-source implementation of HotStuff[4] [39]. We selected this implementation because it implements a Pacemaker [39], contrarily to the implementation used in the original HotStuff paper[5]. Additionally, it provides well-documented benchmarking scripts to measure performance in various conditions, and it is close to a production system (it provides real networking, cryptography, and persistent storage). It is implemented in Rust, uses Tokio[6] for asynchronous networking, ed25519-dalek[7] for elliptic curve based signatures, and data-structures are persisted using RocksDB[8]. It uses TCP to achieve reliable point-to-point channels, necessary to correctly implement the distributed system abstractions.

By default, this HotStuff implementation uses a traditional mempool to disseminate transactions before consensus; we modify its `mempool` crate to use Layered-SMR instead. Its pull protocol simply synchronizes missing blocks by querying the block's creator; we modify its `synchronizer` module to implement the probabilistic pull-deliver protocol described in Section 4.1. We use a rust port of BackBlaze to implement Read-Solomon erasure coding [9] necessary for our push protocol, and traditional Merkle trees as commitment scheme. We use the library smtree[10] operating as traditional Merkle tree (rather than sparse Mekrle tree) as illustrated in library's test-suite[11]. We are open-sourcing Layered-SMR[12] along with any measurements data to enable reproducible results[13].

6.2 Evaluation

We evaluate the throughput and latency of Layered-SMR through experiments on Amazon Web Services (AWS). We then show its improvements over a baseline monolithic HotStuff, called Baseline-HotStuff, with no mempool optimizations[14] (validators simply disseminate transactions as part of the consensus block). We particularly aim to demonstrate that Layered-SMR (C1) drastically improves throughput in the common case (no faulty validators), (C2) the pull protocol (Section 4.1) is efficient in that it does not introduce significant latency overhead, and (C3) drastically improves both latency and throughput in the presence of crash-faults. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open research question [6].

We deploy a testbed on AWS, using `m5.8xlarge` instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Frankfurt (eu-central-1), and Tokyo (ap-northeast-1). Parties are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and run Linux Ubuntu server 20.04. We select these machines because they provide decent performance and are in the price range of 'commodity servers'.

In the rest of this section, each measurement in the graphs is the average of 2 independent runs, and the error bars represent one standard deviation. Our baseline experiment parameters are 10 honest validators, a maximum block size of 500KB, a transaction size of 512B, and one benchmark client per validator (collocated on the same machine) submitting transactions at a fixed rate for a duration of 5 minutes. The leader timeout value is set to 5 seconds.

We experimentally determined that with small committee sizes the coin bias of the pull protocol (see Section 4) does not influence performance. We thus run the experiments of this section with an unbiased coins ($k = 0.5$, see Section 4). With 50% probability the pull protocol either (i) reconstructs the batch by requesting erasure-coded shards from every validator, or (ii) directly requests the batch from $\sqrt{n}$ randomly selected validators (where n is the committee size).

6.2.1 Benchmark in the common case

Figure 3a illustrates the latency and throughput of Layered-SMR and Baseline-HotStuff for varying numbers of validators.

[4] https://github.com/asonnino/hotstuff

[5] https://github.com/hot-stuff/libhotstuff

[6] https://tokio.rs

[7] https://github.com/dalek-cryptography/ed25519-dalek

[8] https://rocksdb.org

[9] https://github.com/rust-rse/reed-solomon-erasure

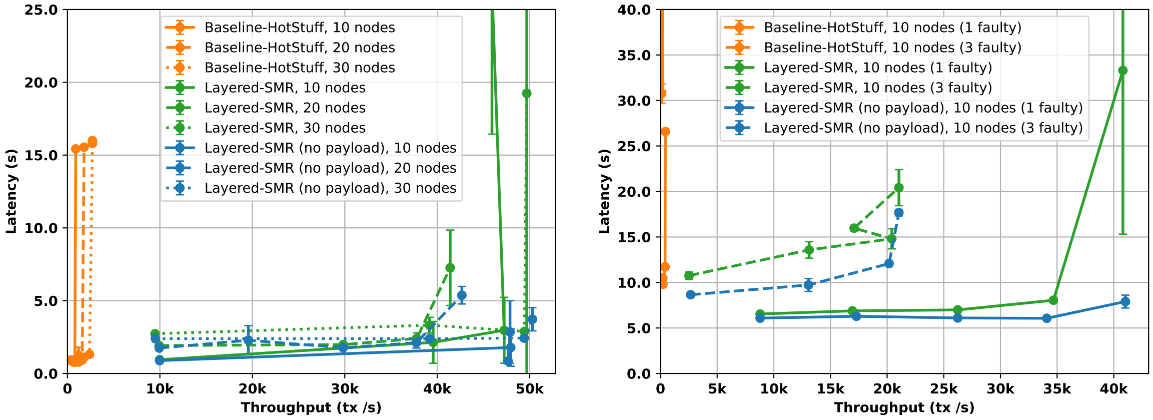
[10] https://github.com/novifinancial/smtree

[11] https://github.com/novifinancial/smtree/blob/17cb9f0c9f949d9f1a134133d76ab7168c6d0b42/src/tests.rs#L259

[12] https://github.com/asonnino/hotstuff/tree/lazy-dolphin

[13] https://github.com/asonnino/hotstuff/tree/lazy-dolphin/data

[14] https://github.com/asonnino/hotstuff/tree/d771d4868db301bcb5e3deaa915b5017220463f6



(a) Measurements with 10, 20, 30 validators. No faulty validators.

(b) Measurements with 10 validators; 0, 1, and 3 crash-faults.

Figure 3: Comparative throughput-latency under crash-faults of Layered-SMR and Baseline-HotStuff. WAN measurements, 500KB maximum block size, and 512B transaction size.

The throughput of Baseline-HotStuff (see Figure 3a, orange lines), with a naive mempool as originally proposed, is quite low. With either 10, 20, or 30 validators throughput never exceeds 2,500 tx/s, although latency at such low throughput is very good at around 1 second. Such surprisingly low numbers are comparable to other works [4], who find HotStuf's performance to be 3,500 tx/s on LAN without modifications such as only transmitting hashes [36]. Performance evaluations [40] of LibraBFT [7] that uses Baseline-HotStuff, report throughput of around 500 tx/s.

Layered-SMR exhibits a significantly higher throughput than Baseline-HotStuff. It remains stable around 40,000 - 50,000 tx/s for a committee of 10, 20 and 30 nodes, making a 20x improvement over Baseline-HotStuff. Figure 3a supports the claim (C1) that Layered-SMR significantly improves the protocol's throughput. Despite its high throughput, Layered-SMR's latency is higher than Baseline-HotStuff, at around 2-3 secs (for all committee sizes). This is expected and caused by the decoupling of transactions dissemination from consensus. Executing the push protocol of Section 4.1 requires erasure-code and cryptographically commit to the shards of transaction batches before making the batch available to consensus. Figure 3a displays two measures of *latency*. The blue lines (labelled 'no payload') measure the time elapsed from when the client submits the transaction to when the transaction is committed by one validator. The green lines measure the time elapsed from when the client submits the transaction to when the transaction is committed by one validator *and the validator retrieved and reconstructed all transaction data*. The blue and green lines are close, thus supporting the claim (C2) that our pull protocol is efficient in that it does not introduce significant latency overhead.

6.2.2 Benchmark under crash-faults

Figure 3b depicts the performance of Layered-SMR and Baseline-HotStuff when a committee of 10 validators suffers 1 to 3 crash-faults (the maximum that can be tolerated in this setting). Baseline-HotStuff suffers a massive degradation in throughput as well as a dramatic increase in latency. For 3 faults, the throughput of Baseline-HotStuff drops by over 20x (dropping to about 130 tx/s) and its latency increases by 30x compared to no faults. In contrast, Layered-SMR maintain a good level of throughput: the underlying push-pull protocol continues collecting and disseminating transactions despite the crash-faults, and is not overly affected by the faulty validators. Layered-SMR's throughput drops from 50,000 to 20,000 when experiencing 3 faults, and its latency increases from 2 secs to 15 secs. The reduction in throughput is in great part due to losing the capacity of faulty validators, and the increase in latency is due to the leader timeout (set to 5 sec). When operating with 3 faults, Layered-SMR provides a 150x throughput increase and about 3x latency reduction with respect to Baseline-HotStuff. Figure 3b support the claim (C3) that Layered-SMR drastically improves both latency and throughput in the presence of crash-faults.

7 Discussion

Recent works have employed AVID in blockchain systems to boost performance (e.g., [30]). However, formalizing the requirements of the PoA&R module in a blockchain architecture has shown that AVID is stronger than necessary. In fact, the exact definition enabled us to propose a solution with retrieval sub-protocols that do not satisfy AVID requirements. These scalable retrieval protocols reduce the expected cost per node in large-scale systems. To support the theoretical analysis of our protocols, we provided extensive simulations as well as real-world experimental results showing that the Layered-SMR performs significantly better than state-of-the-art monolithic approach.

There are several questions that have arisen during this work. One natural direction to consider is more complex choices for $\mathcal{S}$, such as giving higher probability to sampling a process that we have not previously sampled, or randomly choosing k instead of having it fixed a priori. However, it is not obvious how to analyze such stochastic mechanisms. More practical directions to explore are: what choice of PoA&R module best suits a system based on the system's size? Can we use cloud-based solutions for an optimistic and more scalable PoA&R? Finally, while our definition covers some settings, others are left to be defined. For example, what are the properties of PoA&R in a permissionless setting?

On a general note, formally defining modularity in blockchains is an important endeavour. It would facilitate combining contributions from different parts of the community to establish a truly distributed ecosystem.

8 Acknowledgements

The core of this work has been done when the authors where part of the Novi Research group.

References

[1] Daniel J Abadi and Jose M Faleiro. An overview of deterministic database systems. *Communications of the ACM*, 61(9):78–88, 2018.

[2] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

[4] Salem Alqahtani and Murat Demirbas. Bottlenecks in blockchain consensus protocols. *CoRR*, abs/2103.04234, 2021.

[5] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps.* O'reilly Media, 2018.

[6] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. Twins: Bft systems made robust. In *Principles of Distributed Systems*, 2021.

[7] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[8] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2012.

[9] Richard E Blahut. *Theory and practice of error control codes*, volume 126. Addison-Wesley Reading, 1983.

[10] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, 2016.

[11] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[12] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[13] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

[14] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, 1993.

[15] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[16] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[17] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 570–587, 2021.

[18] James Cowling and Barbara Liskov. Granola:{Low-Overhead} distributed transaction coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, 2012.

[19] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

[20] Sourav Das, Zhuolun Xiang, and Ling Ren. Balanced quadratic reliable broadcast and improved asynchronous verifiable information dispersal. *Cryptology ePrint Archive*, 2022.

[21] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[22] Diem. Welcome to the diem project. `https://www.diem.com/en-us/`, 2022.

[23] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5), 2017.

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

[25] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. Scalable byzantine reliable broadcast. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[26] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[27] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 565–574. IEEE, 2000.

[28] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.

[29] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science*, 645:1–24, 2016.

[30] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

[31] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[32] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[33] Kamilla Nazirkhanova, Joachim Neu, and David Tse. Information dispersal with provable retrievability for rollups. *arXiv preprint arXiv:2111.12323*, 2021.

[34] Nym. Building the next generation privacy infrastrucutre. https://nymtech.net.

[35] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 180–194, 2021.

[36] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput BFT for blockchains. *CoRR*, abs/1906.05552, 2019. URL: `http://arxiv.org/abs/1906.05552`, `arXiv:1906.05552`.

[37] Vega. Toward a new era of finance. `https://vega.xyz`, 2022.

[38] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks. *arXiv preprint arXiv:2007.02754*, 2020.

[39] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[40] Jiashuo Zhang, Jianbo Gao, Zhenhao Wu, Wentian Yan, Qize Wu, Qingshan Li, and Zhong Chen. Performance analysis of the libra blockchain: An experimental study. *CoRR*, abs/1912.05241, 2019.

A Pseudo-code for Deterministic Pull-deliver

Algorithm 5: (Deterministic) Linear Pulling Protocol: code for process p_i

```
upon PoA_pull($vc_{sig}, \sigma$) do
    Send($PULL, vc_{sig}, \sigma$) to all
    isPulling←$true$
upon Receiving ($vc_{sig}, c_j, \pi_j$) from $p_j$ for the first time do
    if isPulling $\wedge$ Verify($vc_{sig}, c_j, \pi_j$) then
        CodedVector[$j$] $\leftarrow c_j$
        if $|CodedVector| > f$ then
            $b \leftarrow$ EC.decode(CodedVector)
            if Verify($b, vc_{sig}, CodedVector$) then
                PoA_deliver($b$)
            else
                PoA_deliver($\perp$)
upon Receiving ($PULL, vc_{sig}, \sigma$) from $p_{pull}$ for the first time do
    if Verify($vc_{sig}, \sigma$) then
        if previously received ($vc_{sig}, c_i, \pi_i$) from $p_s$ then
            Send($vc_{sig}, c_i, \pi_i$) to $p_{pull}$
        if did not previously committed $vc_{sig}$ then
            PoA_commit($vc_{sig}, \sigma$)          // unnecessary.  provides reliable BCast in
             addition.
```