

Minimmit: Fast Finality with Even Faster Blocks

BRENDAN KOBAYASHI CHOU¹, ANDREW LEWIS-PYE^{1,2}, and PATRICK O'GRADY¹,

¹ Commonware, USA and ² London School of Economics, UK

Achieving low-latency consensus in geographically distributed systems remains a key challenge for blockchain and distributed database applications. To this end, there has been significant recent interest in State-Machine-Replication (SMR) protocols that achieve *2-round finality* under the assumption that $5f + 1 \leq n$, where n is the number of processors and f bounds the number of processors that may exhibit Byzantine faults. In these protocols, instructions are organised into *views*, each led by a different designated leader, and 2-round finality means that a leader's proposal can be finalised after just a single round of voting, meaning two rounds overall (one round for the proposal and one for voting).

We introduce Minimmit, a Byzantine-fault-tolerant SMR protocol with lower latency than previous 2-round finality approaches. Our key insight is that view progression and transaction finality can operate on different quorum thresholds without compromising safety or liveness. Experiments simulating a globally distributed network of 50 processors, uniformly assigned across ten virtual regions, show that the approach leads to a 23.1% reduction in view latency and a 10.7% reduction in transaction latency compared to the state-of-the-art.

1 Introduction

Protocols for State-Machine-Replication ('blockchain' protocols) are commonly designed to tolerate periods of unreliable message delivery. Formally, this means that protocols are required to satisfy *consistency* (all correct processors agree on the sequence of finalised transactions) and *liveness* (transactions are eventually finalised by correct processors) in the *partially synchronous* setting. In this setting, it is known [13] that satisfying both liveness and consistency is possible precisely if $n \geq 3f + 1$, where n is the number of processors and f bounds the number of processors that may exhibit Byzantine faults.

A key metric when considering the efficiency of State-Machine-Replication (SMR) protocols is *transaction latency*, i.e., the time it takes for transactions to be *finalised*. Most protocols organise operations into *views*, led by designated *leaders*, making the number of communication rounds per view a crucial consideration in analysing latency. Under the standard assumption that $n \geq 3f + 1$, it is known [3] that '3-round' finality is optimal: here 3-round finality means that a leader's proposal can be finalised after two further rounds of voting (this gives three communication rounds overall: round 1 for the proposal and rounds 2 and 3 for voting). Accordingly, standard protocols such as PBFT [10] and Tendermint [7, 8] have 3-round finality. On the other hand, it has been known for some time [26] that 2-round finality (one round for the proposal and one further round of voting) is possible under the assumption that $n \geq 5f + 1$. Recent research has seen a significant renewed interest in approaches to 2-round finality: In the last year, Matter Labs released ChonkyBFT [14], Offchain Labs released Kudzu [30], Anza Labs and Solana released Alpenglow [18], and Supra Research and Espresso Systems released Hydrangea [31]. In part, the motivation for this focus on the $n \geq 5f + 1$ assumption is driven by the scale of modern blockchain systems. When SMR protocols were first deployed in the 1980s, n was small. Today, a typical blockchain system may consist of thousands of processors, meaning that Byzantine attacks on more than a third of participants would be extremely costly, and are deemed unlikely. This makes it appropriate to shift the resilience requirement from $n \geq 3f + 1$ to $n \geq 5f + 1$, since doing so allows for a significant reduction in latency.

This paper describes Minimmit, which differs from previous protocols with 2-round finality by foregoing the *slow path* that is used by most previous protocols when 2-round finality fails, and by allowing processors to proceed to the next view after receiving only $2f + 1$ votes. The idea behind

this tradeoff is that, in a globally distributed network of processors with a range of connection speeds, receiving a smaller quorum of votes will often take significantly less time than waiting for a larger quorum of votes. Requiring the smallest possible quorum for view progression therefore reduces view latency (time between views). This also reduces transaction latency, i.e., the time it takes to finalise transactions, since transactions wait for less time before being included in a block.

1.1 Evidence for reduced latency

To evaluate the performance of Minimmit, we implemented a network simulator¹ that runs custom workloads over configurable network topologies. This simulator provides each participant the opportunity to drive the simulation (i.e. propose a block) while collecting telemetry by region and role. To model realistic network conditions, we apply latency and jitter to message broadcasts using empirical measurements of AWS inter-region performance over the public internet from the past year.

We distinguish *view latency* (time between views), *block latency* (time to finalise blocks), and *transaction latency* (time to finalise transactions). To analyse how different protocols compare across these metrics, we tested Simplex [11], Kudzu [30], and Minimmit on a globally distributed network of 50 processors uniformly assigned across ten “virtual” regions: us-west-1, us-east-1, eu-west-1, ap-northeast-1, eu-north-1, ap-south-1, sa-east-1, eu-central-1, ap-northeast-2, and ap-southeast-2. While we implemented Simplex specifically, finalisation times will be similar for other standard 3-round finality protocols that employ ‘all-to-all’ message sending, such as PBFT and Tendermint, since they use the same structure of leader proposal, followed by two rounds of voting. Likewise, Kudzu is representative of other 2-round finality protocols that employ ‘all-to-all’ message sending and a fast path mechanism, such as Alpenglow² and Hydrangea. Under these conditions, Simplex achieves view latency 194ms (with standard deviation $\sigma=30\text{ms}$) and block latency 299ms ($\sigma=26\text{ms}$). Kudzu achieves view latency 190ms and block latency 220ms ($\sigma=29\text{ms}$). Minimmit achieves view latency 146ms ($\sigma=21\text{ms}$) and block latency 220ms ($\sigma=28\text{ms}$). Minimmit reduces view latency by 25% compared to Simplex and 23% compared to Kudzu. Minimmit reduces block latency by 26% compared to Simplex, equivalent to Kudzu.

For fixed sized blocks, decreased view latency clearly translates into increased throughput. However, it also produces reduced transaction latency for users. Consider a transaction submitted immediately after block construction at ‘height’³ h . This transaction must await inclusion in a block of height $h + 1$ and for that block’s subsequent finalisation. In recently proposed protocols operating under the $n \geq 5f + 1$ assumption—such as Kudzu, Alpenglow, and Hydrangea—view progression requires $n - 2f$ votes. Consequently, transaction finalisation requires waiting for both the completion of the current view (190ms) and the finalisation of the subsequent block containing the transaction (220ms), totalling 410ms in this model. In contrast, Minimmit’s view progression threshold of $2f + 1$ enables the same transaction to achieve finalisation in $146\text{ms} + 220\text{ms} = 366\text{ms}$, representing a 10.7% reduction in end-to-end latency.

1.2 Our contributions

Our contributions are as follows:

- (1) We describe a novel view-change mechanism to reduce view latency;

¹<https://github.com/commonwarexyz/monorepo/tree/19f19d32760daf1d497295726ec92a1e6b84959f/examples/estimator>

²As discussed in Section 7, direct comparisons with Alpenglow are complicated by the fact that Alpenglow incorporates a scheme for disseminating erasure-coded block data during fixed 400ms slots, and so is non-responsive. The fixed 400ms window causes significantly increased view latency for Alpenglow for the parameters considered in our experiments.

³The *height* of a block is its number of ancestors. Ancestors are formally defined in Section 2.

- (2) We give formal proofs of safety, liveness and optimistic responsiveness,⁴ under the $n \geq 5f + 1$ assumption;
- (3) We carry out experimental evaluations, showing roughly a 23% reduction in view latency and an 11% reduction in transaction latency compared to the state-of-the-art.
- (4) In Section 6, we describe (along with other optimisations) a mechanism that uses aggregate signatures to bound the communication required to achieve liveness after periods of asynchrony. This approach may also be applied to improve efficiency for other protocols in the ‘Simplex mould’.

1.3 Dropping the slow path

Many 2-round finality protocols implement a ‘slow path’ to be used when 2-round finality fails: generally, this slow path just requires an extra round of voting. Since Minimmit achieves improved latency by foregoing the ‘slow path’, it is interesting to analyse any resulting sacrifice in resilience, when compared to other leading protocols.

Alpenglow. Alpenglow is formally analysed under the same assumption that $n \geq 5f + 1$. While there is some consideration of circumstances in which the protocol can tolerate a further f crash failures, the required assumptions for this case (essentially that Byzantine leaders cannot carry out a form of proposal equivocation) do not hold under partial synchrony.

Kudzu. Kudzu makes the more general assumption that $n \geq 3f + 2p + 1$ for a tunable parameter p . Of course, in the case that $p = f$, this corresponds to the same assumption that $n \geq 5f + 1$. The protocol guarantees liveness and consistency so long as at most f processors display Byzantine faults. It also guarantees that, during synchrony, a correct leader can finalise a block after a single round of voting, so long as the number of faulty processors is $\leq p$. For a correct leader during synchrony, the ‘slow path’ (requiring two rounds of voting) is therefore only relevant if the number of faulty processors is strictly between p and f . In this case, the quorum required for finality after a single round of voting is also larger than for Minimmit ($n - p$ rather than $n - f$). Kudzu therefore gives no strict ‘like-for-like’ improvement in resilience compared to Minimmit, although the tunable parameter that Kudzu provides is beneficial.

Hydrangea. Compared to Minimmit, Alpenglow, and Kudzu, Hydrangea has improved resilience to crash failures. As before, consider the case of a correct leader during synchrony. For a parameter $k \geq 0$, and for a system of $n = 3f + 2c + k + 1$ processors, Hydrangea achieves finality after 1 round of voting, so long as the number of faulty processors (Byzantine or crash) is at most $p = \lfloor \frac{c+k}{2} \rfloor$. In the case that $c = 0$, this aligns precisely with the bounds provided by Kudzu. However, in more adversarial settings with up to f Byzantine faults and c crash faults, Hydrangea also obtains finality after two rounds of voting. Of course, the benefit of Minimmit is that it achieves reduced transaction latency in the case that $n \geq 5f + 1$.

1.4 Paper structure

The remainder of the paper is structured as follows:

- Section 2 describes the formal setup.
- Section 3 describes the intuition behind Minimmit.
- Section 4 formally specifies the Minimmit protocol.
- Section 5 gives formal proofs of consistency and liveness.

⁴Optimistic responsiveness will be defined in Section 5.3.

- Section 6 describes a number of optimisations, such as the use of threshold signatures, erasure coding, and the use of aggregate signatures to speed up message dissemination and fast recovery after periods of asynchrony.
- Section 7 describes our experiments and results.
- Section 8 describes related work.
- Section 9 contains some final comments and conclusions.

2 The Setup

We consider a set $\Pi = \{p_0, \dots, p_{n-1}\}$ of n processors. For f such that $5f+1 \leq n$, at most f processors may become corrupted by the adversary during the course of the execution, and may then display *Byzantine* (arbitrary) behaviour. Processors that never become corrupted by the adversary are referred to as *correct*.

Cryptographic assumptions. Our cryptographic assumptions are standard for papers on this topic. Processors communicate by point-to-point authenticated channels. We use a cryptographic signature scheme, a public key infrastructure (PKI) to validate signatures, and a collision resistant hash function H .⁵ We assume a computationally bounded adversary. Following a common standard in distributed computing and for simplicity of presentation (to avoid the analysis of negligible error probabilities), we assume these cryptographic schemes are perfect, i.e., we restrict attention to executions in which the adversary is unable to break these cryptographic schemes.

The partial synchrony model. As noted above, processors communicate using point-to-point authenticated channels. We consider the standard partial synchrony model, whereby the execution is divided into discrete timeslots $t \in \mathbb{N}_{\geq 0}$ and a message sent at time t must arrive at time $t' > t$ with $t' \leq \max\{\text{GST}, t\} + \Delta$. While Δ is known, the value of GST is unknown to the protocol. The adversary chooses GST and also message delivery times, subject to the constraints already defined. Correct processors begin the protocol execution before GST and are not assumed to have synchronised clocks. For simplicity, we do assume that the clocks of correct processors all proceed in real time, meaning that if $t' > t$ then the local clock of correct p at time t' is $t' - t$ in advance of its value at time t . Using standard arguments, our protocol and analysis can be extended in a straightforward way to the case in which there is a known upper bound on the difference between the clock speeds of correct processors.

Transactions. Transactions are messages of a distinguished form, signed by the *environment*. Each timeslot, each processor may receive some finite set of transactions directly from the environment. We make the standard assumption that transactions are unique (repeat transactions can be produced using an increasing ‘ticker’ or timestamps [10]).

State machine replication. If σ and τ are sequences, we write $\sigma \preceq \tau$ to denote that σ is a prefix of τ . We say σ and τ are *compatible* if $\sigma \preceq \tau$ or $\tau \preceq \sigma$. If two sequences are not compatible, they are *incompatible*. If σ is a sequence of transactions, we write $\text{tr} \in \sigma$ to denote that the transaction tr belongs to the sequence σ . Each processor p_i is required to maintain an append-only log, denoted \log_i , which at any timeslot is a sequence of distinct transactions. We also write $\log_i(t)$ to denote the value \log_i at the end of timeslot t . The log being append-only means that for $t' > t$, $\log_i(t) \preceq \log_i(t')$. We require the following conditions to hold in every execution:

Consistency. If p_i and p_j are correct, then for any timeslots t and t' , $\log_i(t)$ and $\log_j(t')$ are compatible.

⁵In Section 6, we will also consider optimisations of the protocol that use threshold signatures, aggregate signatures, and erasure codes.

Liveness. If p_i and p_j are correct and if p_i receives the transaction tr then, for some t , $\text{tr} \in \log_j(t)$.

Blocks, parents, ancestors, and finalisation. We specify a protocol that produces *blocks* of transactions. Among other values, each block b specifies a value $b.\text{Tr}$, which is a sequence of transactions. There is a unique *genesis* block b_{gen} , which is considered *finalised* at the start of the protocol execution. Each block b other than the genesis block has a unique *parent*. The *ancestors* of b are b and all ancestors of its parent (while the genesis block has only itself as ancestor), and each block has the genesis block as an ancestor. Each block b thus naturally specifies an extended sequence of transactions, denoted $b.\text{Tr}^*$, given by concatenating the values $b'.\text{Tr}$ for all ancestors b' of b , removing any duplicate transactions. When a processor p_i *finalises* b at timeslot t , this means that, upon obtaining all ancestors of b , it sets $\log_i(t)$ to extend $b.\text{Tr}^*$. Two blocks are *inconsistent* if neither is an ancestor of the other.

3 The intuition

In this section, we give an informal account of the intuition behind the protocol design.

One round of voting. We aim to specify a standard form of *view-based* protocol, in which each view has a designated *leader*. If a view has a correct leader and begins after GST, the leader should send a block b to all other processors, who will then send a signed *vote* for b to all others. Upon receipt of $n - f$ votes for b (by distinct processors), our intention is that processors should then be able to immediately finalise b : this is called *2-round finality* [26] (one round to send the block and one round for voting). $5f - 1 \leq n$ is necessary and sufficient⁶ for 2-round finality: we make the assumption that $5f + 1 \leq n$ for simplicity. We'll call a set of $n - f$ votes for a block an *L-notarisation* (where 'L' stands for 'large').

When to enter the next view? We specified above that a single L-notarisation should suffice for finality. However, as well as achieving 2-round finality, we also wish processors to proceed to the next view immediately upon seeing a smaller set of votes, which we'll call an *M-notarisation* (where the 'M' stands for 'mini'). As explained in Section 1, the rationale behind this is that, in realistic scenarios, receiving $n - f$ votes will generally take much longer than receiving a significantly smaller set of votes (of size $2f + 1$, say). Proceeding to view $v + 1$ immediately upon seeing an M-notarisation for b in view v therefore speeds up the process of block formation: processors can begin view $v + 1$ earlier, and then finalise b upon later receiving the larger notarisation.

How many votes should we require for an M-notarisation? Note that, so long as correct processors don't vote for more than one block in a view, the following will hold:

(X1) If b for view v receives an L-notarisation, then no block $b' \neq b$ for view v receives $2f + 1$ votes.

To see this, suppose towards a contradiction that b for view v receives an L-notarisation and $b' \neq b$ for view v receives $2f + 1$ votes. Let P be the set of processors that contribute to the L-notarisation for b , and let P' be the set of processors that vote for b' . Then $|P \cap P'| \geq (n - f) + (2f + 1) - n = f + 1$. So, $P \cap P'$ contains a correct processor, which contradicts the claim that correct processors don't vote for two blocks in one view.

So, if we set an M-notarisation to be a set of $2f + 1$ votes for b , and allow processors to proceed to view $v + 1$ upon seeing an M-notarisation for b in view v , then any processor receiving an M-notarisation for b knows that no block $b' \neq b$ for view v can receive an L-notarisation. In particular, it is useful to see things from the perspective of the leader, p_i say, of view $v + 1$. If p_i has

⁶See [21] and <https://decentralizedthoughts.github.io/2021-03-03-2-round-bft-smr-with-n-equals-4-f-equals-1/>

seen an M-notarisation for b in view v , and so long as p_i resends this to other processors,⁷ p_i can be sure that all processors have proof that no block other than b could have been finalised in view v . So, p_i can propose a block with b as parent, and all correct processors will have evidence that it is safe to vote for p_i 's proposal.

Nullifications. Since it may be the case that no block for view v receives an M-notarisation (e.g., if the leader is Byzantine), processors must sometimes produce signed messages indicating that they wish to move to view $v + 1$ because of a lack of progress in view v : we'll call these $\text{nullify}(v)$ messages.⁸ We do not yet specify precisely when processors send $\text{nullify}(v)$ messages (we will do so shortly). For now, we promise only that a statement analogous to (X1) will hold for $\text{nullify}(v)$ messages:

(X2) If some block b for view v receives an L-notarisation, then it is not the case that $2f + 1$ processors send $\text{nullify}(v)$ messages.

With the promise of (X2) in place, let's define a *nullification* to be a set of $2f + 1$ $\text{nullify}(v)$ messages (signed by distinct processors). We specify that a processor should enter view $v + 1$ upon receiving either:

- An M-notarisation for view v , or;
- A nullification for view v .

So long as (X2) holds, any processor receiving a nullification for view v knows that no block for view v can receive an L-notarisation.

Defining valid proposals so as to maintain consistency. Suppose p_i is the leader of view v :

- (i) Upon entering view v , p_i finds the greatest $v' < v$ such that it has received an M-notarisation for some block b for view v' : since p_i has entered view v , it must have received nullifications for all views in (v', v) . Processor p_i then proposes a block b' with b as parent.
- (ii) Other processors will vote for b' , so long as they see nullifications for all views in (v', v) and an M-notarisation for b : since all processors (including p_i) will resend notarisations and nullifications upon first receiving them, if view v begins after GST, p_i can therefore be sure that all correct processors will receive the messages they need in order to vote for b' .

It should also not be difficult to see that this will guarantee consistency (see Section 5 for the full proof). Towards a contradiction, suppose that b_1 for view v_1 receives an L-notarisation, and that there is a least view $v_2 \geq v_1$ such that some block b_2 for view v_2 that does not have b_1 as an ancestor receives an M-notarisation. From (X1) it follows that $v_2 > v_1$. By our choice of v_2 , and since correct processors will not vote for blocks until they see an M-notarisation for the parent, it follows that the parent of b_2 must be for a view $< v_1$. This gives a contradiction, because correct processors would not vote for b_2 in view v_2 without seeing a nullification for view v_1 . Such a nullification cannot exist, by (X2).

Ensuring liveness. To ensure liveness, we must first guarantee that if all correct processors enter a view, then they all eventually leave the view. To this end we allow that, while correct processors will not vote in any view v after sending a $\text{nullify}(v)$ message, they *may* send $\text{nullify}(v)$ messages after voting. More precisely, correct p_i will send a $\text{nullify}(v)$ message while in view v if either:

- (a) Their 'timer' for view v expires (time 2Δ passes after entering the view) *before voting*, or;
- (b) They receive messages from $2f + 1$ processors, each of which is either:

⁷To reduce communication complexity, this could be done using a threshold signature scheme, but we defer such considerations to Section 6.

⁸Our approach here is somewhat similar to Simplex [11], but the reader need not have familiarity with that protocol to follow the discussion.

- A $\text{nullify}(v)$ message, or;
- A vote for a view v block different than a view v block that p_i has already voted for.

Combined with the fact that correct processors will forward on nullifications and notarisations upon receiving them, the conditions above achieve two things. First, they suffice to ensure that (X2) is satisfied. If b for view v receives an L-notarisation, then let P be the correct processors that vote for b , let $P' = \Pi \setminus P$, and note that $|P'| \leq 2f$. No processor in P can send a $\text{nullify}(v)$ message via (a) or vote for a view v block other than b . It follows that no processor in P can be caused to send a $\text{nullify}(v)$ message via (b). So, at most $2f$ processors can send $\text{nullify}(v)$ messages.

The conditions above also suffice to ensure that if all correct processors enter a view v , then they all eventually leave the view. Towards a contradiction, suppose all correct processors enter view v , but it is not the case that they all leave the view. Since correct processors forward nullifications and notarisations upon receiving them, this means that no correct processor leaves view v . Each correct processor eventually receives, from at least $n - f$ processors, either a vote for some block for view v or a $\text{nullify}(v)$ message. If any correct processor receives an M-notarisation for the block they voted for, then we reach an immediate contradiction. So, suppose otherwise. If p_i is a correct processor that votes for a view v block, it follows that p_i receives messages from at least $(n - f) - (2f) = n - 3f \geq 2f + 1$ processors, each of which is either a $\text{nullify}(v)$ message or a vote for a view v block different than the view v block that p_i votes for. So, p_i sends a $\text{nullify}(v)$ message via (b). Any correct processor that does not vote for a view v block also sends a $\text{nullify}(v)$ message, so all correct processors send $\text{nullify}(v)$ messages, giving the required contradiction.

Adding an extra instruction to send votes. Once we have ensured that correct processors progress through the views, establishing liveness amounts to showing that each correct leader after GST finalises a new block. This now follows quite easily, using the reasoning outlined in (i) and (ii) above, where we explained that (after GST) processors are guaranteed to receive all the messages they need to verify the validity of a block proposed by a correct leader. However, a subtle issue does require us to stipulate one further context in which correct processors should vote for a block. Suppose view v begins after GST and that the leader for view v is correct. Since a correct processor p_i proceeds to view $v + 1$ immediately upon seeing an M-notarisation for a view v block b , and since it is possible that this is received *before* p_i receives all nullifications required to verify that p_i should vote for b , the possibility apparently remains that correct processors will proceed to view $v + 1$ without b receiving an L-notarisation, i.e., we do not yet have any guarantee that all correct processors will vote for b . To avoid this, we stipulate that, if p_i receives an M-notarisation for b and has not previously sent a $\text{nullify}(v)$ message or voted during view v , then p_i should vote for b before entering view $v + 1$. In this case, the fact that b has already received an M-notarisation means that some correct processors have already voted for b , so it is safe for p_i to do the same. The formal proof appears in Section 5.

Intuition summary (informal):

- In each view, the leader proposes a block and all processors then vote (one round of voting). Correct processors vote for at most one block in each view, which ensures (X1).
- An L-notarisation suffices for finalization, while an M-notarisation or a nullification suffices to move to the next view.
- Processors only vote for the block b for view v , with parent b' for view v' , if they have seen a notarisation for b' and nullifications for all views in (v', v) . Using (X1) and (X2), this suffices for safety.

- To ensure progression through the views, processors send $\text{nullify}(v)$ messages upon timing out (2Δ after entering the view) before voting, or upon receiving proof that no block for view v will receive an L-notarisation. This ensures (X2) is satisfied.
- Once we are given that processors progress through views, liveness follows from the fact that each correct leader after GST will finalise a new block: correct processors will vote for the leader's proposal because the leader themselves will have sent all messages required to verify the validity of the block. Since correct processors also vote for the leader's block in view v upon seeing an M-notarisation for it (if they have not already voted in view v , or sent a $\text{nullify}(v)$ message), this ensures all correct processors vote for the block before leaving view v , and it receives an L-notarisation.

4 The formal specification

We initially give a specification aimed at simplicity. In Section 6, we will describe optimisations, such as the use of threshold signatures to reduce communication complexity. In what follows, we suppose that all messages are signed by the sender, and that, when a correct processor sends a message to 'all processors', it regards that message as immediately received by itself. For the sake of simplicity, we also initially assume (without explicit mention in the pseudocode) that correct processors automatically send new transactions to all others upon first receiving them - we will revisit this assumption in Section 6. The pseudocode uses a number of message types, local variables, functions and procedures, detailed below.

The function $\text{lead}(v)$. The value $\text{lead}(v)$ specifies the leader for view v . To be concrete, we set⁹ $\text{lead}(v) := p_j$, where $j = v \bmod n$.

Blocks. The *genesis block* is the tuple $b_{\text{gen}} := (0, \lambda, \lambda)$, where λ denotes the empty sequence (of length 0). A block other than the genesis block is a tuple $b = (v, \text{Tr}, h)$, signed by $\text{lead}(v)$, where:

- $v \in \mathbb{N}_{\geq 1}$ (thought of as the view corresponding to b);
- Tr is a sequence of distinct transactions;
- h is a hash value (used to specify b 's parent).

We also write $b.\text{view}$, $b.\text{Tr}$ and $b.h$ to denote the corresponding entries of b . If $b.\text{view} = v$, we also refer to b as a 'view v block'.

Votes. A *vote* for the block b is a message of the form (vote, b) .

M-notarisations. An *M-notarisation* for the block b is a set of $2f + 1$ votes for b , each signed by a different processor. (By an *M-notarisation*, we mean an M-notarisation for some block.)

L-notarisations. An *L-notarisation* for the block b is a set of $n - f$ votes for b , each signed by a different processor. We note that, since an L-notarisation is a larger set of votes than an M-notarisation, if p_i has received an L-notarisation for b , then it has necessarily received an M-notarisation for b .

Nullify(v) messages. For $v \in \mathbb{N}_{\geq 1}$, a $\text{nullify}(v)$ message is of the form $(\text{nullify}, v)$.

Nullifications. A *nullification* for view v is a set of $2f + 1$ $\text{nullify}(v)$ messages, each signed by a different processor. (By a *nullification*, we mean a nullification for some view.)

The local variable S . This variable is maintained locally by each processor p_i and stores all messages received. It is considered to be automatically updated, i.e., we do not give explicit instructions in the pseudocode updating S . We say a set of messages M' is *contained in* S if $M' \subseteq S$. We also regard

⁹Of course, leaders can also be randomly selected, if given an appropriate source of common randomness.

S as containing a block b whenever S contains any message (tuple) with b as one of its entries. Initially, S contains only b_{gen} and an L-notarisation (and an M-notarisation) for b_{gen} .

The local variable v . Initially set to 1, this variable specifies the present view of a processor.

The local timer T . Each processor p_i maintains a local timer T , which is initially set to 0 and increments in real-time. (Processors will be explicitly instructed to reset their timer to 0 upon entering a new view.)

The local variables `nullified`, `proposed`, and `notarised`. These are used by p_i to record whether it has yet sent a `nullify(v)` message, whether it has yet proposed a block for view v , and the block it has voted for in the present view: `nullified` and `proposed` are initially set to false, while `notarised` is initially set to \perp (a default value different than any block). These values will be explicitly reset upon entering a new view.

The function `SelectParent(S, v)`. This function is used by the leader of a view, p_i say, to select the parent block to build on. If $v' < v$ is the greatest view¹⁰ such that S contains an M-notarisation for some b with $b.\text{view} = v'$, and if b is the lexicographically least such block, the function outputs b .

The procedure `ProposeChild(b, v)`. This procedure is executed by the leader p_i of view v to determine a new block. To execute the procedure, p_i :

- Forms a sequence of distinct transactions Tr , containing all transactions received by p_i and not included in $b'.\text{Tr}$ for any $b' \in S$ which is an ancestor of b , and;
- Sends the block $(v, \text{Tr}, H(b))$ to all processors.

When S contains a valid proposal for view v . This condition is satisfied when S contains:

- Precisely one block b of the form $b = (v, \text{Tr}, h)$ signed by $\text{lead}(v)$;
- An M-notarisation for some b' with $H(b') = h$, and with $b'.\text{view} = v'$ (say), and;
- A nullification for each view in the open interval (v', v) .

When (i)–(iii) are satisfied w.r.t. b , we say S contains a valid proposal b for view v .

New nullifications and notarisations. Processors will be required to forward on all newly received nullifications and M-notarisations. To make this precise, we must specify what is to count as a ‘new’ nullification/notarisation. At timeslot t , p_i regards a nullification $N \subseteq S$ for some view v (not necessarily equal to v) as *new* if:

- S (as locally defined) did not contain a nullification for view v at any smaller timeslot, and;
- N is lexicographically least amongst nullifications for view v contained in S .

At timeslot t , p_i regards an M-notarisation $Q \subseteq S$ for block b as *new*¹¹ if:

- S did not contain an M-notarisation for b at any smaller timeslot, and;
- Q is lexicographically least amongst M-notarisations for b contained in S .

For ease of reference, message types and local variables are displayed in Tables 1 and 2. The pseudocode appears in Algorithm 1.

¹⁰There must exist such a view, since S always contains an M-notarisation for the genesis block.

¹¹Since it is not necessary for liveness, our pseudocode does not require processors to forward L-notarisations. One could also require processors to forward L-notarisations, and the proofs of Section 5 would go through unchanged.

Message type	Description
b_{gen}	The tuple $(0, \lambda, \lambda)$, where λ is the empty string
block $b \neq b_{\text{gen}}$	A tuple (v, Tr, h) , signed by $\text{lead}(v)$
vote for b	A message (vote, b)
$\text{nullify}(v)$	A message of the form $(\text{nullify}, v)$
nullification for v	A set of $2f + 1$ $\text{nullify}(v)$ messages, each signed by a different processor
M-notarisation for b	A set of $2f + 1$ votes for b , each signed by a different processor
L-notarisation for b	A set of $n - f$ votes for b , each signed by a different processor

Table 1. Messages

Variable	Description
v	Initially 1, specifies the present view
T	Initially 0, a local timer reset upon entering each view
nullified	Initially false, specifies whether already sent $\text{nullify}(v)$ message
proposed	Initially false, specifies whether already proposed a block for view v
notarised	Initially set to \perp , records block voted for in present view
S	Records all received messages, automatically updated
	Initially contains only b_{gen} and M/L-notarisations for b_{gen}

Table 2. Local variables

5 Protocol Analysis

5.1 Consistency

We say block b receives an M-notarisation if $b = b_{\text{gen}}$ or at least $2f + 1$ processors send votes for b . Similarly, we say b receives an L-notarisation if $b = b_{\text{gen}}$ or at least $n - f$ processors send votes for b . View v receives a nullification if at least $2f + 1$ processors send $\text{nullify}(v)$ messages.

LEMMA 5.1 (ONE VOTE PER VIEW). *Correct processors vote for at most one block in each view, i.e., if p_i is correct then, for each $v \in \mathbb{N}_{\geq 1}$, there exists at most one b with $b.\text{view} = v$ such that p_i sends a message (vote, b) .*

PROOF. Recall that \perp is a default value, different than any block. Each correct processor's local value notarised is initially set to \perp , and is also set to \perp upon entering any view (lines 17 and 21). A correct processor p_i will only vote for a block if $\text{notarised} = \perp$ (lines 10 and 20). The claim of the lemma holds because, upon voting for any block b , p_i either sets $\text{notarised} := b$ (line 11) and then does not redefine this value until entering the next view, or else immediately enters the next view (lines 20 and 21). \square

LEMMA 5.2 ((X1) IS SATISFIED). *If b receives an L-notarisation, then no block $b' \neq b$ with $b'.\text{view} = b.\text{view}$ receives an M-notarisation.*

PROOF. Given Lemma 5.1, this now follows as in Section 3. Towards a contradiction, suppose that b receives an L-notarisation and that $b' \neq b$ with $b'.\text{view} = b.\text{view}$ receives an M-notarisation. Let P be the set of processors that contribute to the L-notarisation for b , and let P' be the set of processors that vote for b' . Then $|P \cap P'| \geq (n - f) + (2f + 1) - n = f + 1$. So, $P \cap P'$ contains a correct processor, which contradicts Lemma 5.1. \square

Algorithm 1 : the instructions for p_i

```

1: At every timeslot  $t$ :
2:   Send new nullifications in  $S$  to all processors;           ▷ ‘new’ as defined in Section 4
3:   Send new M-notarisations in  $S$  to all processors;
4:
5:   If  $p_i = \text{lead}(v)$  and proposed = false:
6:     ProposeChild(SelectParent( $S, v$ ),  $v$ );                  ▷ Send out a new block
7:     Set proposed := true;
8:
9:   If  $S$  contains a valid proposal  $b$  for view  $v$ :           ▷ As defined in Section 4
10:    If notarised =  $\perp$  and nullified = false:
11:      Set notarised :=  $b$  and send (vote,  $b$ ) to all processors;    ▷ Send vote
12:
13:    If  $T = 2\Delta$ , nullified = false and notarised =  $\perp$ :
14:      Set nullified := true and send (nullify,  $v$ ) to all processors;  ▷ Send nullify( $v$ )
15:
16:    If  $S$  contains a nullification for  $v$ :
17:      Set  $v := v + 1$ , nullified := false, proposed := false, notarised :=  $\perp$ ,  $T := 0$ ;    ▷ Go to next view
18:
19:    If  $S$  contains an M-notarisation for some  $b$  with  $b.\text{view} = v$ :
20:      If notarised =  $\perp$  and nullified = false, send (vote,  $b$ ) to all processors;  ▷ Send vote
21:      Set  $v := v + 1$ , nullified := false, proposed := false, notarised :=  $\perp$ ,  $T := 0$ ;    ▷ Go to next view
22:
23:
24:    If nullified = false, notarised  $\neq \perp$  and  $S$  contains  $\geq 2f + 1$  messages, each signed by a
25:    different processor, and each either:
26:      (i) A message (nullify,  $v$ ), or;
27:      (ii) Of the form (vote,  $b$ ) for some  $b$  s.t.  $b.\text{view} = v$  and notarised  $\neq b$ :
28:        Set nullified := true and send (nullify,  $v$ ) to all processors;    ▷ Send nullify( $v$ ) message upon proof of no progress for  $v$ 
29:
30:
31:    If  $S$  contains a new L-notarisation for any block  $b$ :
32:      Finalise  $b$ ;                                              ▷ Finalisation as specified in Section 2

```

LEMMA 5.3 ((X2) IS SATISFIED). *If b receives an L-notarisation and $v = b.\text{view}$, then view v does not receive a nullification.*

PROOF. Towards a contradiction, suppose b receives an L-notarisation, $v = b.\text{view}$, and view v receives a nullification. Let P be the correct processors that vote for b , let $P' = \Pi \setminus P$, and note that $|P'| \leq 2f$. Since view v receives a nullification, it follows that some processor in P must send a nullify(v) message. So, let t be the first timeslot at which some processor $p_i \in P$ sends such a message. Since p_i cannot send a nullify(v) message upon timeout (lines 13-14), p_i must send the nullify(v) message at t because the conditions of lines 24-27 hold for p_i at t , i.e., p_i must have received $\geq 2f + 1$ messages, each signed by a different processor, and each of the form:

- (i) (nullify, v), or;
- (ii) (vote, b') for some $b' \neq b$ with $b'.\text{view} = v$.

By Lemma 5.1, no processor in P sends a message of form (ii). By our choice of t , no processor in P sends a message of form (i) prior to t . Combined with the fact that $|P'| \leq 2f$, this gives the required contradiction. \square

LEMMA 5.4 (CONSISTENCY). *The protocol satisfies Consistency.*

PROOF. Towards a contradiction, suppose that two inconsistent blocks, b and b' say, both receive L-notarisations. Without loss of generality suppose $b.\text{view} \leq b'.\text{view}$. Set $b_1 := b$ and $v_1 := b_1.\text{view}$. Then there is a *least* $v_2 \geq v_1$ such that some block b_2 satisfies:

- (1) $b_2.\text{view} = v_2$;
- (2) b_1 is not an ancestor of b_2 , and;
- (3) b_2 receives an M-notarisation.

From Lemma 5.2, it follows that $v_2 > v_1$. According to clause (ii) from the definition of when S contains a valid proposal for view v_2 , correct processors will not vote for b_2 in line 11 until they receive an M-notarisation for its parent, b_0 say. Correct processors will not vote for b_2 in line 20 until b_2 has already received an M-notarisation, meaning that at least $f + 1$ correct processors must first vote for b_2 via line 11, and b_0 must receive an M-notarisation. By our choice of v_2 , it follows that $b_0.\text{view} < v_1$. This gives a contradiction, because, by clause (iii) from the definition of a valid proposal for view v_2 , correct processors would not vote for b_2 in line 11 without receiving a nullification for view v_1 . By Lemma 5.3, such a nullification cannot exist. So, block b_2 cannot receive an M-notarisation (and no correct processor votes for b_2 via either line 11 or 20). \square

5.2 Liveness

LEMMA 5.5 (PROGRESSION THROUGH VIEWS). *Every correct processor enters every view $v \in \mathbb{N}_{\geq 1}$.*

PROOF. Towards a contradiction, suppose that some correct processor p_i enters view v , but never enters view $v + 1$. Note that correct processors only leave any view v' upon receiving either a nullification for the view, or else an M-notarisation for some view v' block. Since correct processors forward new nullifications and notarisations upon receiving them (lines 2 and 3), the fact that p_i enters view v but does not leave it means that:

- All correct processors enter view v ;
- No correct processor leaves view v .

Each correct processor eventually receives, from at least $n - f$ processors, either a vote for some view v block, or a $\text{nullify}(v)$ message. If any correct processor receives an M-notarisation for a view v block, then we reach an immediate contradiction. So, suppose otherwise. If p_j is a correct processor that votes for a view v block b , it follows that p_j receives messages from at least $(n - f) - (2f) = n - 3f \geq 2f + 1$ processors, each of which is either:

- (i) A $\text{nullify}(v)$ message, or;
- (ii) A vote for a view v block different than b .

So, the conditions of lines 24-27 are eventually satisfied, meaning that p_j sends a $\text{nullify}(v)$ message (line 28). Any correct processor that does not vote for a view v block also sends a $\text{nullify}(v)$ message, so all correct processors send $\text{nullify}(v)$ messages. All correct processors therefore receive a nullification for view v and leave the view (line 17), giving the required contradiction. \square

LEMMA 5.6 (CORRECT LEADERS FINALISE BLOCKS). *If $p_i = \text{lead}(v)$ is correct, and if the first correct processor to enter view v does so after GST, then p_i sends a block to all processors and that block receives an L-notarisation.*

PROOF. Suppose $p_i = \text{lead}(v)$ is correct and that the first correct processor p_j to enter view v does so at timeslot $t \geq \text{GST}$. If $v > 1$, processor p_j enters view v upon receiving either a nullification for view $v - 1$, or else an M-notarisation for some view $v - 1$ block. Since p_j forwards on all new notarisations and nullifications that it receives (lines 2 and 3), it follows that all correct processors enter view v by $t + \Delta$ (note that this also holds if $v = 1$). Processor p_i therefore sends a new block b to all processors by $t + \Delta$, which is received by all processors by $t + 2\Delta$. Let b' be the parent of b and suppose $b'.\text{view} = v'$. Then p_i receives an M-notarisation for b' by $t + \Delta$. Since p_i forwards on all new notarisations that it receives (line 3), all correct processors receive an M-notarisation for b' by $t + 2\Delta$. Since p_i has entered view v , it must also have received nullifications for all views in the open interval (v', v) by $t + \Delta$, and all correct processors receive these by $t + 2\Delta$. All correct processors therefore vote for b (by either line 11 or 20)¹² before any correct processor sends a nullify(v) message. The block b therefore receives an L-notarisation, as claimed. \square

LEMMA 5.7 (LIVENESS). *The protocol satisfies Liveness.*

PROOF. Suppose correct p_i receives the transaction tr . Let v be a view with $\text{lead}(v) = p_i$ and such that the first correct processor to enter view v does so after GST. By Lemma 5.6, p_i will send a block b to all processors, and b will receive an L-notarisation. From the definition of the ProposeChild procedure, it follows that tr will be included in $b'.\text{Tr}$ for some ancestor b' of b , and all correct processors will add tr to their log upon receiving all ancestors of b (see the final paragraph of Section 2). Correct processors only vote for blocks whose parent has already received an M-notarisation. All ancestors of b' of b must therefore receive M-notarisations, meaning that at least $f + 1$ correct processors send each such b' to all processors, and correct processors receive all ancestors of b . \square

5.3 Optimistic responsiveness

Let $\delta \leq \Delta$ be the *actual* (unknown) least upper bound on message delay after GST, and let $f_a \leq f$ be the actual (unknown) number of Byzantine processors. If a transaction tr is first received by a correct processor at time t , and is first finalised by all correct processors (i.e., appended to \log_i for every correct p_i) at time $t + \ell$, then we say *latency for* tr is ℓ . We say a protocol is *optimistically responsive* if latency is $O(f_a\Delta + \delta)$ for all transactions that are first received by any correct processor after GST: in particular, this means latency after GST is $O(\delta)$ when processors act correctly. In this section we show that Minimmit is optimistically responsive. In fact, it satisfies the stronger condition that latency after GST is $O(\delta)$ when *leaders* act correctly.¹³

LEMMA 5.8. *Suppose $\text{lead}(v)$ is correct and that the first correct processor to enter view v does so at $t \geq \text{GST}$. Then all correct processors leave view v and finalise a view v block by $t + O(\delta)$.*

PROOF. Proving the claim of the lemma just involves reviewing the proof of Lemma 5.6 and observing that the leader's block will actually be finalised by all correct processors within time $O(\delta)$ of any correct processor entering the view.

As before, suppose first $p_i = \text{lead}(v)$ is correct and that the first correct processor to enter view v does so at timeslot $t \geq \text{GST}$. Since correct processors forward on all new notarisations and nullifications that they receive, it follows that all correct processors enter view v by $t + \delta$. Processor p_i therefore sends a new block b to all processors by $t + \delta$, which is received by all processors by $t + 2\delta$. Let b' be the parent of b and suppose $b'.\text{view} = v'$. Then, from the definition of the function

¹²The point of line 20 is to ensure this part of the proof goes through. As noted in Section 3, without it, there is the possibility that correct processors move to the next view upon seeing an M-notarisation, before they are able to vote via line 11, thereby failing to guarantee an L-notarisation.

¹³Since the notion of leaders is protocol specific, we prefer to use the more general definition as stated, but the stronger result also follows directly from the proofs given in this section.

SelectParent, it follows that p_i receives an M-notarisation for b' by $t + \delta$. Since p_i forwards on all new notarisations that it receives (line 3), all correct processors receive an M-notarisation for b' by $t + 2\delta$. Since p_i has entered view v , it must also have received nullifications for all views in the open interval (v', v) by $t + \delta$, and all correct processors receive these by $t + 2\delta$. All correct processors therefore vote for b (by either line 11 or 20) by $t + 2\delta$, and before any correct processor sends a nullify(v) message. All correct processors therefore receive b together with an L-notarisation (and an M-notarisation) for b by $t + 3\delta$, and also leave view v by this time. This establishes the claim of the lemma. \square

LEMMA 5.9. *Suppose the first correct processor to enter view v does so at $t \geq \text{GST}$. Then, whether or not $\text{lead}(v)$ is correct, all correct processors leave view v by $t + O(\Delta)$.*

PROOF. Suppose the first correct processor to enter view v does so at $t \geq \text{GST}$. Towards a contradiction, suppose some correct processor does not leave view v by $t + 2\Delta + 3\delta$. As before, it follows that all correct processors enter view v by $t + \delta$. By timeslot $t + \delta + 2\Delta$, all correct processors have either voted for some view v block, or else sent a nullify(v) message. If any correct processor receives an M-notarisation for a view v block by $t + 2\Delta + 2\delta$, then it forwards it on to all processors. This means all correct processors leave the view by $t + 2\Delta + 3\delta$, giving an immediate contradiction. So, suppose otherwise. If p_j is a correct processor that votes for a view v block b , it follows that, by $t + 2\Delta + 2\delta$, p_j receives messages from at least $(n - f) - (2f) = n - 3f \geq 2f + 1$ processors, each of which is either:

- (i) A nullify(v) message, or;
- (ii) A vote for a view v block different than b .

So, the conditions of lines 24-27 are satisfied at this time, meaning that p_j sends a nullify(v) message (line 28). Any correct processor that does not vote for a view v block also sends a nullify(v) message by this time. So, all correct processors receive a nullification for view v by $t + 2\Delta + 3\delta$, giving the required contradiction. \square

LEMMA 5.10. *Minimmit is optimistically responsive.*

PROOF. Suppose tr is first received by a correct processor at $t \geq \text{GST}$. Since we assume correct processors send new transactions to all other processors upon first receiving them, tr is received by all correct processors by $t + \delta$. Let v_0 be the greatest view that any correct processor is in at $t + \delta$, and let v_1 be the least view $> v_0$ such that $\text{lead}(v)$ is correct. From Lemmas 5.8 and 5.9, it follows that all correct processors enter view v by time $t + O(f_a\Delta + \delta)$, and that all correct processors also finalise a view v_1 block, b say, by $t + O(f_a\Delta + \delta)$. According to the definition of the procedure $\text{ProposeChild}(b, v)$, tr will be included in an ancestor of b . Since all ancestors of b' of b must receive M-notarisations prior to $\text{lead}(v)$ proposing b , at least $f + 1$ correct processors send each such b' to all processors, and correct processors receive all ancestors of b by $t + O(f_a\Delta + \delta)$. All correct processors therefore finalise tr by time $t + O(f_a\Delta + \delta)$. \square

6 Optimisations

In Section 4, we gave a specification aimed at simplicity. In this section, we describe a number of possible optimisations.

6.1 Progression through views

The specification of Section 4 requires correct processors to progress sequentially through views. To recover quickly from periods of asynchrony, one can allow a correct processor that is presently in view v' to progress immediately to view $v + 1 > v'$ upon seeing a nullification for view v , or an M-notarisation for view v . This requires the following modifications:

- (1) If $v'' < v$ and a correct processor p_i in view v receives an M-notarisation for some view v'' block b , and if p_i has not voted for any view v'' block and has not sent a $\text{nullify}(v'')$ message, then p_i must vote for b . This is now required to ensure that correct leaders finalise new blocks after GST, i.e., that a correct leader's block receives an L-notarisation.
- (2) Upon entering view v , and before running $\text{ProposeChild}(\text{SelectParent}(S, v), v)$, $\text{lead}(v)$ must now wait until there exists $v'' < v$ such that it has received:
 - An M-notarization for some view v'' block, and;
 - Nullifications for all views in (v'', v) .
 Lemmas 5.6 and 5.8 still go through with this change in place, since, if the first correct processor to enter the view does so at $t \geq \text{GST}$, then $\text{lead}(v)$ will still receive these required messages by $t + \delta$.

With these changes in place, the proofs of Sections 5.1–5.3 go through almost unchanged.

6.2 Reducing the size of votes

In Section 4, votes take the form (vote, b) , and so include the entire block b . For constant-sized blocks, this does not affect asymptotic communication complexity. However, when blocks are large, a standard optimisation is to use votes of the form $(\text{vote}, H(b))$, containing only the block's hash.

This optimisation introduces a data availability challenge, which is common to all protocols using votes that only specify the block's hash: a Byzantine leader might propose a block b that receives sufficient votes for finalisation, but fail to send b itself to all correct processors. Since correct processors need the actual block content to update their logs, they must have a mechanism to retrieve missing blocks.

A standard solution exploits the fact that finalisation requires votes from many processors. If a block is finalised, at least $n - f$ processors (including many correct ones) must have received and voted for it. Correct processors can therefore use a (potentially rate-limited) "pull" mechanism to retrieve any missing finalised blocks from peers who possess them, ensuring data availability without relying on potentially Byzantine leaders. An alternative approach, described in Section 6.3, is to have leaders reliably broadcast blocks using erasure coding techniques.

6.3 Erasure coding

In Minimmit, as in all leader-based protocols, leaders must broadcast potentially large blocks to all n processors. With large blocks or high transaction throughput, leader bandwidth can become a bottleneck, limiting overall system performance. Following approaches used in recent SMR protocols [18, 29, 30], we can apply erasure coding techniques [4, 9] to distribute the communication load. The leader encodes each block into n fragments such that any d fragments suffice to reconstruct the original block (where d is a parameter, set as required).

Implementation for Minimmit:

- Set $d = f + 1$ (ensuring availability despite f Byzantine failures).
- Leader sends one unique fragment to each processor.
- Communication overhead: $n(\text{block_size}/d) = n(\text{block_size}/(f + 1)) \approx 5\text{block_size}$.
- Processors vote only after receiving and validating their fragment.
- Each vote includes the processor's fragment to enable block reconstruction.

Correctness guarantee. Since any notarization requires votes from at least $2f + 1$ processors, and at most f are Byzantine, at least $f + 1$ correct processors must contribute to each notarisation, meaning that they send their fragment to all processors. This ensures sufficient fragments are available for all processors to reconstruct any notarised block.

Trade-offs. This optimisation reduces leader bandwidth requirements but adds fragment verification overhead. The approach is most beneficial when block sizes are large relative to network capacity. As noted in Section 6.2, a benefit of the approach is that it ensures correct processors receive all finalised blocks, without requiring the use of a "pull" mechanism to retrieve any missing finalised blocks from peers who possess them.

6.4 Threshold signatures and communication complexity

Section 4 assumes correct processors send newly received transactions to all others. In practice, transactions are typically disseminated through gossip networks, where each processor forwards transactions to a small, constant number of peers. If transactions are constant-bounded in size, this approach maintains constant communication overhead per processor per transaction. Alternative dissemination mechanisms like Narwhal [12] can also be employed. In our analysis here, we treat transaction propagation as a black box and focus on the consensus layer. We assume blocks have constant-bounded size and, following standard practice, analyse only the communication costs required for consensus, taking mempool formation as given.

Message complexity. Per view, the protocol requires:

- Leader proposal: $O(n)$ messages.
- Votes: $O(n^2)$ messages (each processor sends ≤ 1 vote to all others).
- Nullify messages: $O(n^2)$ messages.
- Forwarding notarisations/nullifications: $O(n^2)$ messages.

Communication complexity. Each notarisation and nullification contains $\Omega(n)$ signatures, resulting in communication complexity greater than $O(n^2)$ per view. The standard approach of using threshold signatures [5, 28] can be used to ensure that the communication complexity per view is $O(n^2)$. We redefine:

- An 'M-notarisation' for b to be threshold signature (of constant-bounded length for a given security parameter) formed from $2f + 1$ votes for b by different processors.
- A 'nullification' for view v , to be a threshold signature formed from $2f + 1$ $\text{nullify}(v)$ messages by different processors.

Threshold signature implemetation details. If p_i has not already formed or received an M-notarisation for b , then, upon receipt of $2f + 1$ votes for b by different processors, p_i combines the $2f + 1$ signatures to form an M-notarisation (a single threshold signature). Rather than storing and forwarding the $2f + 1$ votes, p_i stores and forwards the M-notarisation. Similarly, p_i stores and sends nullifications, rather than storing and sending large collections of $\text{nullify}(v)$ messages. Since partial signatures for any processor can be derived from $2f + 1$ partial signatures, we must now stipulate that processors finalise a block b upon receiving $n - f$ votes for b directly from the corresponding processors.

We note that the pseudocode specified in Algorithm 1 does not require processors to forward L-notarisations. If one wished to implement threshold signatures also for L-notarisations (requiring the different threshold $n - f$), then one would need to establish two separate threshold signature schemes (two shared secrets), i.e., we require a separate shared secret for each threshold. This also means that each vote requires two signatures: one corresponding to each threshold. Of course, these two signatures can be computed/verified in parallel. Verification for the two signatures can also be transformed into an aggregate signature verification because the two signatures are over the same message payload.

Alternative approaches. We note that some protocols (e.g., Hotstuff [34]) achieve linear communication complexity per view by relaying all messages via the leader. However, this approach

significantly increases the number of communication rounds required, and the leader anyway acts as a communication bottleneck (e.g., see [24]).

6.5 Compressed nullifications

The problem: nullification build-up during asynchrony. During extended periods of asynchrony, processors may generate nullifications for many consecutive views without being able to finalise new blocks. When synchrony is restored, correct processors must exchange all accumulated nullifications before they can vote for new proposals. Since a processor requires nullifications for all intermediate views between a block’s parent and the current view (see Section 4), this can create substantial communication overhead.

Solution: nullification aggregation. For signature schemes supporting aggregation (e.g., BLS [5]), we can compress consecutive nullifications. Given threshold signatures for views in the range $[v, v']$, we aggregate them into a single signature σ of constant size and send the tuple (v, v', σ) .

Verification. Aggregate signature verification requires specifying the message sequence and corresponding public keys. Here, the message sequence is implicitly defined by the view range $[v, v']$, and each nullification uses the same shared public key from the threshold scheme.

7 Experiments

We test Minimmit against Simplex [11] and Kudzu [30] using a deterministic simulator that executes protocol specifications on configurable network topologies. The simulator implementation and experiment configurations are released under both MIT and Apache-2 licenses.¹⁴ A step-by-step guide for recreating every experiment appears in Appendix A.

Protocol selection. Simplex introduced state-of-the-art transaction latency among 3-round finality protocols. Its design inspired Minimmit: both protocols decouple view iteration from finalisation, shrinking the worst-case delay before a transaction is included in a block. This structural similarity makes it a natural baseline against which to quantify the impact of Minimmit’s relaxed Byzantine fault tolerance.

Kudzu (and the similarly constructed Alpenglow [18]) recently delivered state-of-the-art transaction latency for 2-round finality protocols by concurrently evaluating fast and slow paths. Unlike Alpenglow, which incorporates a scheme for disseminating erasure-coded block data during fixed 400ms slots¹⁵, Kudzu is responsive and serves as a better candidate for comparison to Minimmit. We defer a comprehensive comparison to Alpenglow’s block dissemination to future work.

Simulation procedure. Each simulation iteration designates a leader and treats the remaining processors as replicas. The leader initiates the view by broadcasting a payload of configurable size to all replicas. Replicas process received payloads after at most a configurable number of pending messages and broadcast their own protocol-specific messages. The simulator records the time at which each processor reaches salient protocol milestones and reports the mean and standard deviation over all processors. Because the runtime is deterministic, these measurements are reproducible by re-running the same command.¹⁶

¹⁴<https://github.com/commonwarexyz/monorepo/tree/19f19d32760daf1d497295726ec92a1e6b84959f/examples/estimator>

¹⁵A processor in Votor will not cast a vote for some block until it has recovered the entire block from Rotor.

¹⁶<https://github.com/commonwarexyz/monorepo/tree/19f19d32760daf1d497295726ec92a1e6b84959f/runtime/src/deterministic.rs>

Latency and bandwidth model. For every message transmission we sample a delay from a normal distribution with mean equal to the p50¹⁷ AWS inter-region latency and standard deviation (p90 – p50).¹⁸ Each processor has symmetric 1 Gbps (125,000,000 B/s) ingress and egress budgets, so message delivery time is the sum of the sampled network delay and the bandwidth-limited transmission time. We assume linear bandwidth usage and max-min fairness for bandwidth allocation, yielding conservative latency estimates. In practice, burstable bandwidth and traffic prioritisation provided by cloud operators would only decrease the reported latencies.

Uniform global deployment. We place 50 processors uniformly across ten AWS regions (us-west-1, us-east-1, eu-west-1, ap-northeast-1, eu-north-1, ap-south-1, sa-east-1, eu-central-1, ap-northeast-2, ap-southeast-2) and enable the “reducing the size of votes” optimisation from Section 6.2 for all protocols, so that the leader distributes full blocks (i.e. no erasure coding) while replicas vote on digests. We assume instantaneous block production and begin broadcasting at time $t = 0$. Table 3 summarises the resulting latency when proposing 32 KB blocks, the block size at which Minimmit processes 1,000 transactions per second¹⁹ in this configuration.

Table 3. Uniform global deployment (50 processors, symmetric 1 Gbps links).

Protocol	View Latency	Block Latency	Transaction Latency
Simplex	194.61 ± 30.34 ms	299.34 ± 25.98 ms	493.95 ± 7.50 ms
Kudzu	189.94 ± 29.32 ms	220.31 ± 28.58 ms	410.25 ± 7.61 ms
Minimmit	146.07 ± 21.33 ms	220.3 ± 28.57 ms	366.37 ± 7.06 ms

Minimmit reduces transaction latency by 25.8% relative to Simplex and by 10.7% relative to Kudzu. The improvement stems from Minimmit’s ability to progress views after collecting $2f + 1$ votes.

Region-centric deployment. We next cluster 25 processors in the United States (13 in us-west-1 and 12 in us-east-1) and place the remaining 25 uniformly across the other eight regions. Table 4 reports the resulting latencies for the same 32 KB blocks, increasing Minimmit’s effective processing rate from 1,000 to 1,500 transactions per second at a lower transaction latency.

Table 4. Region-centric deployment (50 processors, symmetric 1 Gbps links).

Protocol	View Latency	Block Latency	Transaction Latency
Simplex	149.95 ± 24.58 ms	222.32 ± 24.05 ms	372.27 ± 6.97 ms
Kudzu	139.77 ± 26.17 ms	185.16 ± 24.09 ms	324.93 ± 7.09 ms
Minimmit	104.93 ± 34.71 ms	187.67 ± 27.13 ms	292.6 ± 7.86 ms

Under this regional skew, Minimmit decreases transaction latency by 21.4% relative to Simplex and by 9.95% relative to Kudzu. Kudzu attains slightly lower block latency because its slow path races the fast path, and in this topology some leaders complete two rounds gathering $3f + 1$ votes before Minimmit collects $n - f$ votes.

Large blocks. Larger blocks take longer to transmit: broadcasting a 1 MB block to 50 processors over 1 Gbps links already requires roughly 400 ms of egress latency.²⁰ Table 5 reproduces the global

¹⁷https://www.cloudping.co/api/latencies?percentile=p_50&timeframe=1Y

¹⁸https://www.cloudping.co/api/latencies?percentile=p_90&timeframe=1Y

¹⁹ Assumes each transaction is 200 B.

²⁰ Recall, we assume bandwidth allocation is max-min fair.

deployment with 1 MB blocks, the block size at which Minimmit processes 10,000 transactions per second²¹ in this configuration.

Table 5. Uniform global deployment (50 processors, symmetric 1 Gbps links).

Protocol	View Latency	Block Latency	Transaction Latency
Simplex	593.61 ± 30.34 ms	698.34 ± 25.98 ms	1291.95 ± 7.50 ms
Kudzu	588.94 ± 29.32 ms	619.31 ± 28.58 ms	1208.25 ± 7.61 ms
Minimmit	545.07 ± 21.33 ms	619.3 ± 28.57 ms	1164.37 ± 7.06 ms

Introducing a naïve Reed–Solomon erasure coding scheme reduces this bottleneck. We split each block into 50 fragments, have the leader broadcast a fragment to every processor, and require replicas to broadcast their fragment to all other processors when casting a vote. The fragment size is determined by the quorum needed for view progression: Simplex targets $3f + 1$ replicas and therefore transmits 61.72 KB fragments so that any $f + 1$ fragments suffice for reconstruction; Minimmit targets $5f + 1$ replicas and sends 104.86 KB fragments to ensure $f + 1$ fragments suffice; Kudzu also targets $5f + 1$ replicas but requires 55.19 KB fragments to guarantee reconstruction from $2f + 1$ fragments. Table 6 reports the resulting latencies, increasing Minimmit’s effective processing rate from 10,000 to 25,000 transactions per second at a lower transaction latency.

Table 6. Uniform global deployment with erasure coding (1 MB blocks).

Protocol	View Latency	Block Latency	Transaction Latency
Simplex	230.62 ± 30.33 ms	335.36 ± 25.97 ms	565.98 ± 7.50 ms
Kudzu	220.94 ± 29.33 ms	251.29 ± 28.63 ms	472.23 ± 7.61 ms
Minimmit	216.0 ± 21.46 ms	290.32 ± 28.52 ms	506.28 ± 7.07 ms

With coding enabled, Minimmit achieves the lowest view latency but not the lowest transaction latency, despite each processor only emitting a single vote message per view (rather than driving concurrent paths). This result is a product of the smaller fragment size required by Kudzu (47.37% smaller than Minimmit’s), which offsets Minimmit’s lower threshold for view progression. With the lowest view latency, Minimmit still *maximises throughput* across all protocols, since blocks of a fixed size are produced at a faster rate.

Our current implementation of Minimmit, however, uses conservative erasure coding parameters. Since Minimmit requires only $2f + 1$ messages to progress through views, we could potentially require $2f + 1$ valid fragments for view progression and design the erasure coding to allow reconstruction from exactly $2f + 1$ fragments rather than the $f + 1$ fragments currently required. This would reduce fragment sizes by approximately 50%, yielding lower transaction latency than Kudzu by combining reduced communication overhead with our faster view progression mechanism. We leave coding parameter optimization for future work.

Increasing per processor bandwidth from 1 Gbps to 10 Gbps under this naïve coding scheme, as illustrated in Table 7, restores Minimmit’s outright transaction latency advantage over Kudzu under this workload.

With more bandwidth to compensate for the larger fragment sizes, Minimmit decreases transaction latency by 25.0% relative to Simplex and 9.07% relative to Kudzu. Increasing bandwidth and reducing latency in this standardised environment, as illustrated in Table 7 and Table 4, consistently

²¹ Assumes each transaction is 200 B.

Table 7. Uniform global deployment with increased bandwidth and erasure coding (10 Gbps per processor and 1 MB blocks).

Protocol	View Latency	Block Latency	Transaction Latency
Simplex	186.63 ± 30.35 ms	291.33 ± 26.0 ms	477.96 ± 7.51 ms
Kudzu	181.86 ± 29.31 ms	212.3 ± 28.6 ms	394.16 ± 7.61 ms
Minimmit	142.14 ± 21.36 ms	216.27 ± 28.58 ms	358.41 ± 7.07 ms

improves Minimmit’s performance relative to Simplex and Kudzu. This observation supports the primary insight employed by Minimmit (and Simplex): faster-than-finality view progression is a fundamental building block for minimising transaction latency.

8 Related work

Classical Byzantine Consensus. The study of protocols for reaching consensus in the presence of Byzantine faults was introduced by Lamport, Shostak, and Pease [22]. Dwork, Lynch and Stockmeyer [13] showed that $n \geq 3f + 1$ is optimal for partial synchrony. Standard protocols using the assumption $n \geq 3f + 1$, such as PBFT [10] and Tendermint [7, 8], satisfy 3-round finality. As shown by [3], this is optimal.

Optimistic Responsiveness. While many standard protocols, such as PBFT, satisfy forms of optimistic responsiveness, a specific form of the concept was first discussed in [27]. Optimistic responsiveness has been further studied in a number of papers (e.g., [2, 34]) and can be defined in a number of ways [23, 25, 34].

Fast-Path Approaches. A long line of work [6, 15, 16, 20, 26, 32] considers protocols with a ‘fast path’, which allows for quick termination/finalisation in certain ‘good’ scenarios. Kursawe [20] describes an agreement protocol that runs with $3f + 1$ processors, but is able to commit in two steps when all processes act correctly and the network is synchronous, falling back to a randomised asynchronous consensus protocol otherwise. FaB [26] extends Kursawe’s approach by more closely integrating the fast path and the fall-back mechanism (the ‘slow path’), and by introducing a parameterised model with $n \geq 3f + 2p + 1$ processors. Fast termination is achieved, so long as at most p processors are Byzantine. Unfortunately, the protocol suffers from a liveness bug [1]. FaB also gives a proof that the assumed bound $n \geq 3f + 2p + 1$ is tight. However, it was pointed out in [21] and [3] that the proof only applies to a specific form of protocol. Kuznetsov et al. [21] show that, in fact, the bound $n \geq 3f + 2p - 1$ is optimal. Zyzzyva [19] also builds on FaB by (similarly) integrating the fast and slow paths, and by describing an SMR protocol, rather than a protocol for one-shot consensus. As pointed out in [1], the view-change mechanism in Zyzzyva does not guarantee safety when leaders are faulty. SBFT [17] also builds on the ideas introduced in FaB in order to allow the fast path to tolerate a small number of crash failures.

Modern 2-Round Protocols. Alpenglow [18] is formally analysed under the assumption that $n \geq 5f + 1$. The paper also considers circumstances in which the protocol can tolerate a further f crash failures, but the required assumptions for this case (essentially that Byzantine leaders cannot carry out a form of proposal equivocation) do not hold under partial synchrony. Banyan [33] carries out the fast path in parallel with the slow path mechanism, but can suffer from unbounded message complexity with faulty leaders. Kudzu (like FaB) makes the more general assumption that $n \geq 3f + 2p + 1$ for a tunable parameter p . Kudzu and Alpenglow also describe the use of erasure coding techniques to allow for significantly improved maximum throughput (such techniques were also employed by SMR protocols in earlier papers, such as [29]).

Compared to Minimmit, Alpenglow, and Kudzu, Hydrangea has improved resilience to crash failures. For a parameter $k \geq 0$, and for a system of $n = 3f + 2c + k + 1$ processors, Hydrangea achieves 2-round finality, so long as the number of faulty processors (Byzantine or crash) is at most $p = \lfloor \frac{c+k}{2} \rfloor$. In the case that $c = 0$, this aligns precisely with the bounds provided by Kudzu. However, in more adversarial settings with up to f Byzantine faults and c crash faults, Hydrangea also obtains finality after two rounds of voting.

As for Minimmit, ChonkyBFT [14] assumes $n \geq 5f + 1$ and employs a single round of voting, but does not have Minimmit’s mechanism for fast view progression.

Positioning of Minimmit. Minimmit assumes $n \geq 5f + 1$ and achieves 2-round finality. The advantage of Minimmit over all previous approaches to 2-round finality is its fast view change mechanism, which, as described in Sections 1 and 7, allows for decreased view and transaction latency. While subjective, we also believe that the simplicity of the protocol will make it attractive to practitioners.

9 Final comments

We have presented Minimmit, a Byzantine fault-tolerant SMR protocol that achieves reduced transaction latency through a novel view-change mechanism. By decoupling view progression from transaction finality—requiring only $2f + 1$ votes for view changes while requiring $n - f$ votes for finalisation—Minimmit demonstrates that significant latency improvements are possible without sacrificing safety or liveness guarantees.

Our experimental evaluation shows an approximately 23% reduction in view latency and an 11% reduction in transaction latency compared to existing approaches, achieved through faster view progression in geographically distributed networks. The protocol’s simplicity, requiring no complex slow-path mechanisms, may facilitate practical adoption in systems where low latency is critical.

10 Acknowledgements

We would like to thank Ittai Abraham, Benjamin Chan, Denis Kolegov, Ling Ren, and Victor Shoup for helpful conversations.

References

- [1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367* (2017).
- [2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 106–118.
- [3] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 331–341.
- [4] Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. 2021. Succinct erasure coding proof systems. *Cryptology ePrint Archive* (2021).
- [5] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.
- [6] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. 2001. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*. Springer, 42–50.
- [7] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation.
- [8] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938* (2018).
- [9] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS’05)*. IEEE, 191–201.
- [10] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.

- [11] Benjamin Y Chan and Rafael Pass. 2023. Simplex consensus: A simple and fast consensus protocol. In *Theory of Cryptography Conference*. Springer, 452–479.
- [12] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.
- [13] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (1988), 288–323.
- [14] Bruno Fran  a, Denis Kolegov, Igor Konnov, and Grzegorz Prusak. 2025. ChonkyBFT: Consensus Protocol of ZKsync. *arXiv preprint arXiv:2503.15380* (2025).
- [15] Roy Friedman, Achour Mostefaoui, and Michel Raynal. 2005. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing* 2, 1 (2005), 46–56.
- [16] Rachid Guerraoui and Marko Vukoli  . 2007. Refined quorum systems. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 119–128.
- [17] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.
- [18] Quentin Kniep, Jakub Sliwinski, and Roger Wattenhofer. 2025. Solana Alpenglow Consensus. <https://www.scribd.com/document/895233790/Solana-Alpenglow-White-Paper> (2025).
- [19] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.
- [20] Klaus Kursawe. 2002. Optimistic byzantine agreement. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings*. IEEE, 262–267.
- [21] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. 2021. Revisiting optimal resilience of fast byzantine consensus. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 343–353.
- [22] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.
- [23] Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. 2024. Lumiere: Making optimal bft for partial synchrony practical. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*. 135–144.
- [24] Andrew Lewis-Pye, Kartik Nayak, and Nibesh Shrestha. 2025. The Pipes Model for Latency Analysis. *Cryptology ePrint Archive* (2025).
- [25] Andrew Lewis-Pye and Tim Roughgarden. 2023. Permissionless Consensus. *arXiv preprint arXiv:2304.14701* (2023).
- [26] J-P Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.
- [27] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29–May 3, 2018 Proceedings, Part II* 37. Springer, 3–33.
- [28] Victor Shoup. 2000. Practical threshold signatures. In *International conference on the theory and applications of cryptographic techniques*. Springer, 207–220.
- [29] Victor Shoup. 2023. Sing a song of Simplex. *Cryptology ePrint Archive* (2023).
- [30] Victor Shoup, Jakub Sliwinski, and Yann Vonlanthen. 2025. Kudu: Fast and Simple High-Throughput BFT. *arXiv preprint arXiv:2505.08771* (2025).
- [31] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2025. Hydrangea: Optimistic Two-Round Partial Synchrony with One-Third Fault Resilience. *Cryptology ePrint Archive* (2025).
- [32] Yee Jiun Song and Robbert Van Renesse. 2008. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*. Springer, 438–450.
- [33] Yann Vonlanthen, Jakub Sliwinski, Massimo Albarello, and Roger Wattenhofer. 2024. Banyan: Fast rotating leader bft. In *Proceedings of the 25th International Middleware Conference*. 494–507.
- [34] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

A Reproducing Experiments

All experiments were executed with the commonware-estimator²². Each run combines one of the protocol scripts below with a network distribution. The workflow is:

²²<https://github.com/commonwarexyz/monorepo/tree/19f19d32760daf1d497295726ec92a1e6b84959f/examples/estimator>

- (1) Save the desired script as <protocol>.lazy.
- (2) Invoke `commonware-estimator -distribution <distribution> <protocol>.lazy` using one of the configurations listed at the end of this section.

Protocol scripts

The following schedules apply to experiments without erasure coding. Set <proposal_bytes> to 32768 for the 32 KB runs and to 1048576 for the 1 MB runs.

Simplex

```
# Simplex
propose{0, size=<proposal_bytes>}
wait{0, threshold=1}
broadcast{1, size=40}
wait{1, threshold=67%}
broadcast{2, size=40}
wait{2, threshold=67%}
```

Kudzu (no coding)

```
# Kudzu (no coding)
propose{0, size=<proposal_bytes>}
wait{0, threshold=1}
broadcast{1, size=40}
wait{1, threshold=61%}
broadcast{2, size=40}
wait{1, threshold=81%} || wait{2, threshold=61%}
```

Minimmit

```
# Minimmit
propose{0, size=<proposal_bytes>}
wait{0, threshold=1}
broadcast{1, size=40}
wait{1, threshold=41%}
wait{1, threshold=81%}
```

Use the erasure-coded variants below for Section 7's coded experiments (message sizes already include shard data).

Simplex (erasure coded)

```
# Simplex (erasure coded)
propose{0, size=61682}
wait{0, threshold=1}
broadcast{1, size=61722}
wait{1, threshold=67%}
broadcast{2, size=40}
wait{2, threshold=67%}
```

Kudzu (erasure coded)

```
# Kudzu (erasure coded)
propose{0, size=55190}
wait{0, threshold=1}
broadcast{1, size=55230}
wait{1, threshold=61%}
broadcast{2, size=40}
wait{1, threshold=81%} || wait{2, threshold=61%}
```

Minimmit (erasure coded)

```
# Minimmit (erasure coded)
propose{0, size=104858}
wait{0, threshold=1}
broadcast{1, size=104898}
wait{1, threshold=41%}
wait{1, threshold=81%}
```

Network distributions

Pair the scripts with one of the following network distributions when calling `commonware-estimator`. The Uniform global distribution is reused for both block sizes; only `<proposal_bytes>` changes.

Uniform global (1 Gbps links)

```
commonware-estimator --distribution \
us-west-1:5:125000000,us-east-1:5:125000000, \
eu-west-1:5:125000000,ap-northeast-1:5:125000000, \
eu-north-1:5:125000000,ap-south-1:5:125000000, \
sa-east-1:5:125000000,eu-central-1:5:125000000, \
ap-northeast-2:5:125000000,ap-southeast-2:5:125000000 \
<protocol>.lazy
```

Region-centric (1 Gbps links)

```
commonware-estimator --distribution \
us-west-1:13:125000000,us-east-1:12:125000000, \
eu-west-1:3:125000000,ap-northeast-1:4:125000000, \
eu-north-1:3:125000000,ap-south-1:3:125000000, \
sa-east-1:3:125000000,eu-central-1:3:125000000, \
ap-northeast-2:3:125000000,ap-southeast-2:3:125000000 \
<protocol>.lazy
```

Uniform global (10 Gbps links)

```
commonware-estimator --distribution \
us-west-1:5:125000000,us-east-1:5:125000000, \
eu-west-1:5:125000000,ap-northeast-1:5:125000000, \
eu-north-1:5:125000000,ap-south-1:5:125000000, \
sa-east-1:5:125000000,eu-central-1:5:125000000, \
ap-northeast-2:5:125000000,ap-southeast-2:5:125000000 \
<protocol>.lazy
```