Sing a song of Simplex*

Victor Shoup[1]

Offchain Labs
victor@shoup.net

April 23, 2024

Abstract. We flesh out some details of the recently proposed Simplex atomic broadcast protocol, and modify it so that leaders disperse blocks in a more communication-efficient fashion. The resulting protocol, called *DispersedSimplex*, maintains the simplicity and excellent — indeed, optimal — latency characteristics of the original Simplex protocol. We also present several variations, including a variant that supports "stable leaders", variants that incorporate very recently developed data dissemination techniques that allow us to disperse blocks even more efficiently, and variants that are "signature free". We also suggest a number of practical optimizations and provide concrete performance estimates that take into account not just network latency but also network bandwidth limitations and computational costs. Based on these estimates, we argue that despite its simplicity, DispersedSimplex should, in principle, perform in practice as well as or better than any other state-of-the-art atomic broadcast protocol, at least in terms of common-case throughput and latency.

1 Introduction

Byzantine fault tolerance (BFT) is the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of some of its components while still functioning properly as a whole. One approach to achieving BFT is via state machine replication [Sch90]: the logic of the system is replicated across a number of machines, each of which maintains state, and updates its state is by executing a sequence of transactions. In order to ensure that the non-faulty machines end up in the same state, they must each deterministically execute the same sequence of transactions. This is achieved by using a protocol for *atomic broadcast*.

In an atomic broadcast protocol, we have a committee of n parties, some of which are honest (and follow the protocol), and some of which are corrupt (and may behave arbitrarily). Roughly speaking, such an atomic broadcast protocol allows the honest parties to schedule a sequence of transactions in a consistent way, so that each honest party schedules the same transactions in the same order. Each party receives various transactions as input — these inputs are received incrementally over time, not all at once. It may be required that a transaction satisfy some type of validity condition, which can be verified locally by each party. These details are application specific and will not be further discussed. Each party outputs an ordered sequence of transactions — these outputs are generated incrementally, not all at once. One key security property of any secure atomic broadcast protocol is **safety**, which means that each party outputs the same sequence of

* This paper has been evolving since first submitted to https://eprint.iacr.org/2023/1916 on Dec. 13, 2023 under the original title of "DispersedSimplex: simple and efficient atomic broadcast". Since then, number of topics have been added: (i) practical optimizations and concrete performance estimates that take into account not just network latency but also network bandwidth limitations and computational costs, (ii) a stable leader variant, (iii) variants based on improved reliable data dissemination techniques recently introduced in [Loc24,LS24], (iv) signature-free variants, and (v) more extensive comparison with other protocols.

transactions. Another key property of any secure atomic broadcast protocol is **liveness**. There are different notions of liveness one can consider, but the basic idea is that the protocol should not get stuck and stop outputting transactions.

Different protocols make different assumptions about the latency guarantees of the network and the number of corrupt parties. Here, we assume that the number of corrupt parties is less than $n/3$, and we consider protocols that are guaranteed to provide safety without any latency assumption, and that are guaranteed to provide liveness only in intervals of "network synchrony", in which the latency is below a certain defined threshold. This is the *partial synchrony* model, introduced in [DLS88]. The bound of $n/3$ on the number of corrupt parties is optimal in this model. Many quite practical atomic broadcast protocols have been proposed in this model, starting with the classic PBFT protocol [CL99], and this is still an area of active research.

In this paper, we consider the recently proposed Simplex atomic broadcast protocol [CP23]. Like many other recent protocols in this space (such as HotStuff [YMR⁺18] and HotStuff-2 [MN23]), Simplex is a leader-based, permissioned blockchain protocol: the protocol proceeds in slots (a.k.a., views, rounds), so that in each slot a leader proposes a block of transactions, and these blocks get added to a tree of blocks. Over time, a path of committed blocks in this tree emerges — safety ensures that all parties agree on the same path of committed blocks. In these protocols, leaders typically are rotated in each slot — either in a round-robin fashion or using some pseudo-random sequence — which also has the nice effect of mitigating against censorship of transactions. The protocol relies on authenticated communication links and a PKI to support digital signatures (preferably aggregate or threshold signatures for better communication complexity).

Simplex is a wonderfully simple, efficient, and elegant protocol. In this paper, we add to the Simplex story in a number of ways:

- We flesh out some missing (but crucial) details of the Simplex protocol that are needed to get a protocol with acceptable communication complexity. Along the way, we make a few other simplifications; in particular, we observe that while the Simplex protocol as specified in [CP23] relies on hash-based chaining of blocks, this turns out to be unnecessary.
- More importantly, we modify the protocol so that leaders disperse blocks in a more communication-efficient fashion, while maintaining its simplicity and excellent — indeed, optimal — latency characteristics. We call this variation on the Simplex protocol **DispersedSimplex**.
- We give a detailed analysis of DispersedSimplex (safety, liveness, and performance), and discuss a number of important implementation details, arguing — based on concrete micro-benchmarks and realistic assumptions on network behavior — that despite its simplicity, in typical scenarios, DispersedSimplex should perform quite well in practice, even for $n \approx 100$.
- We present and analyze a variant of DispersedSimplex that supports "stable leaders" (the paper [CP23] did not investigate such a variant). We argue that this variant can achieve even better performance, mainly because a stable leader can drive the protocol at a significantly faster rate than a constantly rotating leader. The mechanism for failing over from an unresponsive leader is very simple and lightweight (no more complicated or expensive than rotating leaders as in the basic version of the protocol).
- We present and analyze variations of DispersedSimplex (for both rotating and stable leader variants) that incorporate the recently developed techniques of [Loc24,LS24] for improving the communication complexity of reliable broadcast using better data dissemination techniques. We

show how to adapt these techniques to improve the communication complexity of DispersedSimplex even further, without sacrificing latency.
- Perhaps more of theoretical interest, we show how all of the above variations of DispersedSimplex can be implemented without any signatures at all, with roughly the same communication complexity as if aggregate or threshold signatures were used, but somewhat higher latency.
- We compare DispersedSimplex to other protocols in the literature. As we will argue, DispersedSimplex, especially the variants mentioned above that combine stable leaders and better data dissemination techniques, should perform as well as or better than any other state-of-the-art atomic broadcast protocol (including leader-based protocols such as HotStuff [YMR+18] and HotStuff-2 [MN23], as well as DAG-based protocols such as [SAGL23]), at least in terms of common-case throughput and latency. Again, these arguments are based on concrete micro-benchmarks and assumptions on network behavior, and they suggest that it would be worthwhile to measure the actual performance of a well-engineered implementation (which we have not done).

The rest of the paper. Section 2 gives an entirely self contained description of DispersedSimplex — certainly, no knowledge of the Simplex protocol itself is assumed, but some familiarity with similar protocols (like PBFT or HotStuff) may be helpful. Section 3 gives a complete analysis of DispersedSimplex, including proofs of safety and liveness, as well as complexity estimates. This section also contains practical optimizations and concrete performance estimates that take into account not just network latency but also network bandwidth limitations and computational costs. Section 4 briefly presents some minor implementation details and simple variations of DispersedSimplex. Section 5 describes and analyzes a variant of DispersedSimplex that supports stable leaders, and discusses its impact on concrete performance estimates. Section 6 briefly shows how to adapt the recently developed data dissemination techniques of [Loc24,LS24] to DispersedSimplex, and discusses their impact on concrete performance estimates. Section 7 shows how all of the above variations of DispersedSimplex can be implemented without any signatures at all. Section 8 closes with a comparison to other atomic broadcast protocols.

2 The DispersedSimplex protocol

Like many other protocols in this area, the Simplex protocol iterates through slots (a.k.a., views, rounds), where in each slot there is a designated leader who proposes a new block, which is chained to a parent block, and two rounds of voting are used to commit the block. Moreover, to improve latency, the protocol is "pipelined", in the sense that it optimistically moves onto the next slot as soon as the first round of voting succeeds, before the block for that slot is committed. Leaders may be rotated in each slot, either in a round-robin fashion or using some pseudo-random sequence. The DispersedSimplex protocol has the same structure as the Simplex protocol; however, instead of broadcasting the block directly, the slot leader uses well-known techniques for information dispersal to disseminate large blocks in a way that keeps the overall communication complexity low and avoids a bandwidth bottleneck at the leader. In particular, the communication is *balanced*, meaning that each party, including the leader, transmits roughly the same about of data over the network. We will show how the information dispersal can be interleaved with the proposal phase and the first voting round so that no extra latency is incurred.

2.1 Preliminaries

We have a committee of n parties, $P_1, \ldots, P_n$, at most $t < n/3$ of which are corrupt. We assume the parties are connected by authenticated point-to-point channels.

We will not generally assume network synchrony. However, we say the network is *δ-synchronous over an interval* $[a, b + \delta]$ if every message sent from an honest party P at time $t \leq b$ to an honest party Q is received by Q before time $t + \delta$. In this case, for all $t \in [a, b]$, we say that the network is *δ-synchronous at time t*.

2.1.1 Signatures.

We make use of an $(n - t)$-out-of-n threshold signature scheme (although later, in Section 7, we discuss how to avoid signatures with concomitant tradeoffs). We refer to a *signature share* and a *signature certificate*: signature shares from $n - t$ on a given message may be combined to form a signature certificate on that message. This can be implemented as just a set of signatures, or as an aggregate signature scheme (such as one based on BLS signatures [BLS01] as in [BDN18]) or as a threshold version of an ordinary signature scheme (such as one again based on BLS signatures as in [Bol03]). The second and third implementations will result in much more compact threshold signatures. The third implementation requires a set-up phase to distribute shares of a signing key; however, this set-up can be implemented using an atomic broadcast protocol (such as DispersedSimplex) using one of the first two implementations, so that only a PKI set-up is required; once this set-up phase is complete, the protocol can shift to using the third implementation.

The security property for such a threshold signature scheme may be stated as follows.

Quorum Size Property: It is infeasible to produce a signature certificate on a message m, unless $n - t - t'$ honest parties have issued signature shares on m, where $t' \leq t$ is the number of corrupt parties.

Under our assumption that the number of corrupt parties is strictly less than $n/3$, one can easily establish the following standard property.

Quorum Intersection Property: It is infeasible to produce signature certificates on two distinct messages m and m', unless at least one honest party issued signature shares on both m and m'.

2.1.2 Information dispersal.

We explicitly make use of well-known techniques for *asynchronous verifiable information dispersal (AVID)* techniques involving erasure codes and Merkle trees (introduced in [CT05]).

Erasure codes. For integer parameters $k \geq d \geq 1$, a (k, d)-*erasure code* encodes a bit string M as a vector of k *fragments*, $f_1, \ldots, f_k$, in such a way that any d such fragments may be used to efficiently reconstruct M. Note that for variable-length M, the reconstruction algorithm also takes as input the length β of M. The reconstruction algorithm may fail (for example, a formatting error)—if it fails it returns $\perp$, while if it succeeds it returns a message that when re-encoded will yield k fragments that agree with the original subset of d fragments. We assume that all fragments have the same size, which is determined as a function of k, d, and β.

Using a Reed-Solomon code, which is based on polynomial interpolation, we can realize a (k, d)-erasure code so that if $|M| = \beta$, then each fragment has size $\approx \beta/d$. More precisely, using a Reed-Solomon code over binary finite fields, we can always construct a code such that fragments are of size at most $\max(\lceil \beta/d \rceil, \lceil \log_2(k) \rceil)$—the term $\lceil \log_2(k) \rceil$ comes from the fact that we need to work with a field of cardinality at least k. In what follows, we will use the more general upper bound of

$$\beta/d + O(\log(k))$$

on fragment size, which serves as an upper bound for the above construction, as well as for other constructions and implementations (which may impose additional restrictions on the length of fragments, such as being a multiple of some specific constant).

In our protocol, the payload of block will be encoded using an $(n, n-2t)$-erasure code. Such an erasure code encodes a payload M as a vector of fragments $f_1, \ldots, f_n$, any $n-2t$ of which can be used to reconstruct M. This leads to a data expansion rate of (at most) roughly 3; that is, $\sum_i |f_i| \approx n/(n-2t) \cdot |M| < 3|M|$.

Merkle trees. Recall that a Merkle tree allows one party P to commit to a vector of values $(v_1, \ldots, v_k)$ using a collision-resistant hash function by building a (full) binary tree whose leaves are the hashes of $v_1, \ldots, v_k$, and where each internal node of the tree is the hash of its two children. The root r of the tree is the commitment. Party P may "open" the commitment at a position $i \in [k]$ by revealing v_i along with a "validation path" π_i, which consists of the siblings of all nodes along the path in the tree from the hash of v_i to the root r. We call π_i a *validation path from the root under r to the value v_i at position i.* Such a validation path is checked by recomputing the nodes along the corresponding path in the tree, and verifying that the recomputed root is equal to the given commitment r. The collision resistance of the hash function ensures that P cannot open the commitment to two different values at a given position.

Encoding and decoding. For a given payload M of length β, we will encode M as a vector of fragments $(f_1, \ldots, f_m)$ using the $(n, n-2t)$-erasure code, and then form a Merkle tree with root r whose leaves are the hashes of $f_1, \ldots, f_n$. We define the *tag* $\tau := (\beta, r)$.

For a tag $\tau = (\beta, r)$, we shall call (f_i, π_i) a *certified fragment for τ at position i* if

- f_i has the correct length of a fragment for a message of length β, and
- π_i is a correct validation path validation path from the root under r to the fragment f_i at position i.

The function *Encode* takes as input a payload M. It builds a Merkle tree for M as above with root r (encoding M as a vector of fragments, and then building the Merkle tree whose leaves are the hashes of all of these fragments). It returns

$$(\ \tau,\ \{(f_i, \pi_i)\}_{i \in [n]}\ ),$$

where τ is the tag (β, r), β is the length of M, and each (f_i, π_i) is a certified fragment for τ at position i.

The function *Decode* takes as input

$$(\ \tau,\ \{\{(f_i, \pi_i)\}_{i \in \mathcal{I}}\ ),$$

where $\tau = (\beta, r)$ is a tag, $\mathcal{I}$ is a subset of $[n]$ of size $n - 2t$, and each (f_i, π_i) is a certified fragment for τ at position i. It first reconstructs a message M' from the fragments $\{f_i\}_{i \in \mathcal{I}}$, using the size parameter β. If $M' = \perp$, it returns $\perp$. Otherwise, it encodes M' as a vector of fragments $(f'_1, \ldots, f'_n)$ and Merkle tree with root r' from $(f'_1, \ldots, f'_n)$. If $r' \neq r$, it returns $\perp$. Otherwise, it returns M'.

Under collision resistance for the hash function used for the Merkle trees, any $n - 2t$ certified fragments for given tag τ will decode to the same payload — moreover, if τ is the output of the encoding function, these fragments will decode to M (and therefore, if the decoding function outputs $\perp$, we can be sure that τ was maliciously constructed) This observation is the basis for the protocols in [DW20,LLTW20,YPA+21]. Moreover, with this approach, we do not need to use anything like an "erasure code proof system" (as in [ADVZ21]), which would add significant computational complexity (and in particular, the erasure coding would have to be done using parameters compatible with the proof system, which would likely lead to much less efficient encoding and decoding algorithms).

2.2 Protocol data objects

2.2.1 Blocks. A block B is of the form $\mathsf{Block}(v, v', \tau)$, where

- $v = 1, 2, \ldots$ is the slot number associated with the block (and we say B *is a block for slot* v),
- $v' < v$ is the slot number of B's parent block ($v' = 0$ if B's parent is a notional "genesis" block), and
- τ is a tag obtained by encoding B's payload M.

For simplicity, we call a certified fragment for the tag τ a *certified fragment for* B.

2.2.2 Support, commit, and complaint shares and certificates. A *support share from party* P_i *on block* B is an object of the form $\mathsf{SuppShare}(B, \sigma_i, f_i, \pi_i)$, where σ_i is a valid signature share from P_i on the object $\mathsf{Supp}(B)$, and (f_i, π_i) is a certified fragment for B at position i. A *support certificate on* B is an object of the form $\mathsf{SuppCert}(B, \sigma)$, where σ is a valid signature certificate on the object $\mathsf{Supp}(B)$.

A *commit share from party* P_i *on slot* v is an object of the form $\mathsf{CommitShare}(v, \sigma_i)$, where σ_i is a valid signature share from P_i on the object $\mathsf{Commit}(v)$. A *commit certificate on* v is an object of the form $\mathsf{CommitCert}(v, \sigma)$, where σ is a valid signature certificate on the object $\mathsf{Commit}(v)$.

A *complaint share from party* P_i *on slot* v is an object of the form $\mathsf{ComplaintShare}(v, \sigma_i)$, where σ_i is a valid signature share from P_i on the object $\mathsf{Complaint}(v)$. A *complaint certificate on* v is an object of the form $\mathsf{ComplaintCert}(v, \sigma)$, where σ is a valid signature certificate on the object $\mathsf{Complaint}(v)$.

2.3 Subprotocols

We describe our protocol in terms of a main protocol and a few simple subprotocols. In our presentation, these subprotocols are all running concurrently with each other and with the main protocol: a single party can be thought of as running a local instance of the main protocol and each of the subprotocols on different threads on the same CPU. However, this particular architecture is mainly intended just for ease of presentation.

We describe first the data structures and logic of the subprotocols.

2.3.1 Certificate pool. Each party maintains a *certificate pool*. Whenever a party receives a quorum of $n - t$ support, commit, or complaint shares, and it does not already have a corresponding certificate, it will generate a certificate, add it to the pool, and broadcast the certificate to all parties.

Similarly, whenever a party receives a support, commit, or complaint certificate, and it does not already have a corresponding certificate, it will add it to the pool, and broadcast the certificate to all parties.

2.3.2 Complete block tree. Each party also maintains an *complete block tree*. The complete block tree always contains a tree of blocks, rooted at a notional genesis block at slot 0. A block $B = \mathsf{Block}(v, v', \tau)$ is added to the pool if

- $v' = 0$ or the complete block tree contains a parent block $B' = \mathsf{Block}(v', \cdot, \cdot)$,
- the certificate pool contains a support certificate for B;
- the party has received a quorum of $n - 2t$ support shares for B, from which the party can reconstruct the effective payload M of B as

$$M \leftarrow \mathit{Decode}(\tau, \{(f_i, \pi_i)\}_{i \in \mathcal{I}}),$$

  where $\{(f_i, \pi_i)\}_{i \in \mathcal{I}}$ is the corresponding collection of certified fragments for τ;
- $M \neq \perp$ and satisfies some correctness predicate that may depend of the path of blocks (and their payloads) from genesis to block B'.

Unlike the certificate pool, when a party adds a block to its complete block tree, it does not broadcast anything to other parties.

Under cryptographic assumptions, we will see that for any given slot number $v > 0$, there will never be more than one block $B = \mathsf{Block}(v, \cdot, \cdot)$ for slot v in the complete block tree.

2.3.3 Block commitment. We say that a block B for slot v is *explicitly committed by* P if the complete block tree of P contains B and the certificate pool of P contains a commit certificate for slot v. In this case, we say that all of the predecessors of block B in the complete block tree are *implicitly committed by* P. The notional genesis block is always considered to be a committed block. The payloads of committed blocks may be then transmitted in order to the "execution layer" of the protocol stack of a replicated state machine.

2.4 The main protocol

The logic of the main protocol for a party P_j is described in Fig. 1. In the description, $\mathsf{leader}(v)$ denotes the leader for slot v — as discussed above, leaders may be rotated in each slot, either in a round-robin fashion or using some pseudo-random sequence. The details for generating and validating block proposals are described below. In the main protocol, a party makes its decisions based on the objects it its certificate pools and its complete block tree (which are maintained as described in Section 2.3) and the objects it has received from other parties over authenticated channels. The core of the protocol is expressed as in terms of a "wait until either" statement which triggers one of several clauses based different preconditions. Although not strictly necessary, for concreteness, we assume that if more than one clause's precondition is satisfied, then the syntactically first such clause is triggered.

Note that a party will not issue a commit share in a slot if it has already issued a complaint share in that slot — this rule is essential for safety. Also note that a party may issue a support share in a slot even if it has already issued a complaint share in that slot — this rule is not essential and the protocol would also provide both safety and liveness if a party chose not to issue a support share in this case.

DispersedSimplex: *main loop for party P_j*

```
v_last ← 0
for v = 1, 2, ...
    t_start ← clock()
    done ← proposed ← supported ← complained ← false
    while not done do
        wait until either:
            the certificate pool contains a complaint certificate for slot v ⇒
                done ← true
            the complete block tree contains a block for slot v ⇒
                if not complained then broadcast a commit share for v
                done ← true, v_last ← v
            not complained and clock() > t_start + Δ_timeout ⇒
                complained ← true
                broadcast a complaint share for slot v
            leader(v) = P_j and not proposed ⇒
                proposed ← true
(*)             generate block proposal material B, (f_1, π_1), ..., (f_n, π_n)
                for i ∈ [n]: send BlockProp(B, f_i, π_i) to P_i
(**)        not supported and received from leader(v) a valid block proposal BlockProp(B, f_j, π_j) ⇒
                supported ← true
                generate a signature share σ_j on Supp(B)
                broadcast the support share SuppShare(B, σ_j, f_j, π_j)
```

Fig. 1. Logic for main loop of DispersedSimplex protocol for party P_j

2.4.1 Generating block proposals. The logic for generating block proposal material B, $(f_1, \pi_1), \ldots, (f_n, \pi_n)$ in slot v at line $(*)$ is as follows:

- build a payload M that validly extends the path in the complete block tree ending at the block for slot v_{last};
- compute
$$(\tau, \{(f_i, \pi_i)\}_{i \in [n]}) \leftarrow \mathit{Encode}(M);$$
- set $B \coloneqq \mathsf{Block}(v, v_{\text{last}}, \tau)$.

2.4.2 Validating block proposals. To check if $\mathsf{BlockProp}(B, f_j, \pi_j)$ is a valid block proposal from the leader in slot v at line $(**)$, party P_j checks that each of the following conditions holds:

- B is of the form $\mathsf{Block}(v, v', \tau)$, where $v' < v$ and the complete block tree contains a block for slot v';
- the certificate pool contains complaint certificates for slots $v' + 1, \ldots, v - 1$;
- (f_j, π_j) is a certified fragment for τ at position j.

Note that even if some of the conditions do not hold at a given point in time, they may hold at a later point in time. When party P_j sees a block proposal in slot v, it can check the stated conditions — if these conditions fail due to the lack of either a parent block in the complete block tree or a complaint certificate, these conditions will need to be rechecked whenever a new block is added to the complete block tree or a new complaint certificate is added to the certificate pool. We will discuss below (in Section 4.1) how to efficiently implement the test that the certificate pool contains the necessary complaint certificates using a data structure whose size is proportional to the gap between current slot and the last committed slot so that the amortized cost of these tests is $O(1)$ per slot.

3 Analysis

By abuse of terminology, we state security properties unconditionally — they implicitly assume the security of the threshold signature scheme and the collision resistance of the hash functions used to build Merkle trees, and should be understood to hold with all but negligible probability for all efficient adversaries.

3.1 Initial observations

We state some basic properties:

Uniqueness and Validity Property: Suppose that a block B for some slot v is added to the complete block tree of some party. Then no other block for slot v can be added to the complete block tree of that party or any other party. Moreover, if the leader for slot v is honest, B must have been proposed by that leader.

The first part follows from the Quorum Intersection Property, based on the fact an honest party issues a support share for at most one block per slot. The second part follows from the Quorum Size Property.

Completeness Property: If an object X appears in the certificate pool (so X is a support, commit, or complaint certificate) or in the complete block tree (so X is a block), then X (or its equivalent) will eventually appear in the corresponding pool/tree of every other party.[1] Moreover, if X appears in a party's pool/tree at a time t at which the network is δ-synchronous, it will appear in every party's pool/tree before time $t + \delta$.

For the support, commit, and complaint certificates, this is clear. For the blocks in the complete block tree, we are relying on the Quorum Size Property: when a support certificate for a block B is added to the support pool, at least $n - 2t$ honest parties must have already broadcast support shares for B, which contain B as well as fragments sufficient to reconstruct B's payload.

Incompatibility of Complaint and Commit Property: It is impossible to produce both a complaint and commit certificate for the same slot v.

This follows from the Quorum Intersection Property, based on the fact that in each slot, an honest party will never issue both a complaint share and a commit share.

3.2 Safety

Safety follows immediately from the following lemma.

Lemma 3.1 (Safety). *Suppose a party P explicitly commits a block B for slot v, and a block C for slot $w \geq v$ is in the complete block tree of some party Q. Then B is an ancestor of C in Q's complete block tree.*

[1] Note that the "or equivalent" qualification is necessary to account for signature certificates, if these are not necessarily unique.

Proof. By the Incompatibility of Complaint and Commit Property, no complaint certificate for slot v can be produced. Let C' be the parent of C and suppose w' is the slot number of C'. Since C' is in Q's complete block tree, a support certificate for C' must have been produced, which means at least one honest party must have issued a support share for C', which means $v \leq w' < w$. The inequality $v \leq w'$ follows from the fact that there is no complaint certificate for slot v, and an honest party will issue a support share for C only if it has complaint certificates for slots $w' + 1, \ldots, w - 1$.

If $v = w'$, we are done by the (first part of the) Uniqueness and Validity Property, and if $v < w'$, we can repeat the argument inductively with C' in place of C. □

3.3 Liveness

Liveness follows immediately from the following lemmas. The first lemma analyzes the optimistic case where the network is synchronous and the leader of a given slot is honest, showing that the leader's block will be committed.

Lemma 3.2 (Liveness I). *Consider a particular slot $v \geq 1$ and suppose the leader for slot v is an honest party Q. Suppose that the first honest party P to enter the loop iteration for slot v does so at time t. Further suppose that the network is δ-synchronous over the interval $[t, t+3\delta]$ for some δ with $\Delta_{\text{timeout}} \geq 3\delta$. Then each honest party will finish the loop iteration before time $t + 3\delta$ by adding Q's proposed block B to its complete block tree. and will eventually commit B. Moreover, each honest party will eventually commit B, and this will happen before time $t + 4\delta$ if the network remains δ-synchronous over the interval $[t, t + 4\delta]$.*

Proof. By the Completeness Property, before time $t + \delta$, each honest party will enter the loop iteration for slot v by time $t + \delta$, having either a complaint certificate for slot $v - 1$ or a block for slot $v - 1$ in its complete block tree. So before time $t + \delta$, the leader Q will propose a block B that extends a block B' with slot number $v' < v$. By the logic of the protocol, we know that Q must have complaint certificates for slots $v' + 1, \ldots, v - 1$ at the time it makes its proposal. Again by the Completeness Property, before time $t + 2\delta$, each honest party will have B' in its complete block tree and all of these complaint certificates in its certificate pool, and moreover, will receive Q's proposal before this time, and hence will broadcast a support share for Q's proposal by this time. Therefore, before time $t + 3\delta$, each honest party will have added B to its complete block tree. By the assumption that $\Delta_{\text{timeout}} \geq 3\delta$, when each honest party adds B to its complete block tree, the complaint condition will not have been met, and therefore, each honest party will issue a commit share for v at this time. If the network remains δ-synchronous, the commit shares will be received by all honest parties before time $t + 4\delta$. □

The second lemma analyzes the pessimistic case, when the network is asynchronous or the leader of a given round is corrupt. It says that eventually, all honest parties will move on to the next round.

Lemma 3.3 (Liveness II). *Suppose that the network is δ-synchronous over an interval $[t, t + \Delta_{\text{timeout}} + 2\delta]$, for an arbitrary value of δ, and that at time t, some honest party is in the loop iteration for slot v and all other honest parties are in a loop iteration for v or a previous slot. Then before time $t + \Delta_{\text{timeout}} + 2\delta$, all honest parties finish the loop iteration for slot v.*

Proof. By the Completeness Property, every honest party will enter the loop iteration for slot v before time $t + \delta$. By time $t + \delta + \Delta_{\text{timeout}}$, every honest party will have either added a block for slot v to its complete block tree or broadcast a complaint share for slot v. In either case, less than δ time units later all honest parties will have finished the loop iteration for slot v. □

Finally, we note that in periods of asynchrony, for any slot v in which the leader Q is honest, if any block is committed in slot v, it must have been the block proposed by Q. This follows from the (second part of the) Uniqueness and Validity Property.

3.4 Complexity estimates

3.4.1 Communication complexity. We measure the communication complexity per slot. This is the sum over all honest parties P and all parties Q of the bit-length of all slot-v-specific messages sent from P to Q.

The communication complexity per slot of DispersedSimplex is easily seen to be bounded by

$$3n\beta + O(n^2(\kappa + \lambda \log n)),$$

where

- β is a bound on the size of a block,
- κ is a bound on the size of a threshold signature share or certificate,
- and λ is a bound on the size of the hash function outputs used for Merkle trees.

Indeed, the cost breaks down as follows:

- $3n\beta + O(n^2 \log n)$ for disseminating payload fragments,
- $O(n^2 \log n \cdot \lambda)$ for disseminating Merkle paths,
- $O(n^2\kappa)$ for disseminating signature shares and certificates.

If blocks are large, in particular, if $\beta \gg n(\kappa + \lambda \log n)$, the communication complexity will be dominated by the cost of disseminating the payload fragments.

Moreover, the communication load is *balanced*, meaning that each party, *including the leader for a slot*, transmits roughly the same about of data over the network. In fact, as we described the protocol, for large β, each non-leader transmits about 3β bits in total, while the leader transmits about 6β bits in total. In Section 3.5, we discuss a simple variation in which the leader also transmits only about 3β bits. In Section 6, we discuss a variation in which each party transmits only 1.5β–2β bits.

3.4.2 Latency. We may also measure various notions of latency. We define:

- *optimistic proposal-commit latency:* assuming the leader is honest, and that the network is appropriately synchronous, the time it takes for the leader's proposal to be committed by all honest parties (same as the notion of "proposal confirmation time" in [CP23]);
- *optimistic consecutive-proposal latency:* assuming two consecutive leaders are honest, and that the network is appropriately synchronous, the amount of time that elapses between when they make their respective proposals (similar to the notion of "optimistic block time" in [CP23]).

If a given transaction is submitted to the system (i.e., to all parties), the sum of these two latencies upper bounds the total time it takes for a transaction to be included in a proposal and then committed. The optimistic consecutive-proposal latency also upper bounds what we might call the *optimistic reciprocal block throughput*, the reciprocal of the rate at which blocks are proposed (and committed) in a steady state where all leaders are honest and the network is appropriately synchronous.

For DispersedSimplex, just as for Simplex, we readily see that if the network is δ-synchronous with $\Delta_{\text{timeout}} \geq 3\delta$, then the optimistic proposal-commit latency is 3δ and the optimistic consecutive-proposal latency is 2δ. This proposal-commit latency is optimal [ANRX21].

It is also useful to look at the latency between proposals made between non-consecutive honest leaders. That is, if leaders in slots v and $v + k + 1$ are honest, but the k leaders in the intervening slots are crashed or corrupt, how much time may elapse between the time the leader in slot v makes its proposal and the time the leader in slot $v+k+1$ makes its proposal. Let us call this the *optimistic k-gap proposal latency*. For DispersedSimplex, just as for Simplex, this is $2\delta + k \cdot (\Delta_{\text{timeout}} + \delta)$. If leaders are chosen at random, then the probability that there is a gap of size k between slots with honest leaders decreases exponentially with k.

We note that DispersedSimplex protocol is *optimistically responsive*, meaning that it runs as fast as the network will allow so long as leaders are honest.

3.5 Other costs and concrete estimates

The above analysis abstracts away a number of practically important details. Indeed, our latency estimates in Section 3.4.2 only took into account propagation delays caused by network latency, but did not take into account transmission delay (caused by limited network bandwidth) and computation delay (caused by limited compute bandwidth).

In this section, we discuss other costs and make some concrete estimates for performance under specific assumptions. We are generally interested in values of n up to around 100, where each of the n parties is running commodity hardware and connected to a WAN with typical network bandwidth and latency.

We first consider the computational cost of erasure coding. This should not have a significant impact on the overall system performance, assuming one uses a reasonably good implementation of erasure coding algorithms. One such implementation is the `reed-solomon-simd` library at `https://github.com/AndersTrier/reed-solomon-simd`, which is based on [LC12,LAHC16]. We benchmarked this implementation with parameters corresponding to $t = 32$ and $n = 3t + 1 = 97$ and payload sizes of 100KB and 1MB on a Macbook Pro with an Apple M1 Max CPU. The encoder runs at a rate of nearly 2GB/s for both payload sizes. The decoder runs at a rate of about 250MB/s for the 100KB payload and about 500MB/s per second for the 1MB payload. Generally, the encoder speed is independent of the payload size and the decoder speed increases with the payload size (because fixed costs get amortized). At these speeds, it is very unlikely that the erasure coding will be a bottleneck.

We next consider the computational cost of signature generation, verification, and aggregation. Let us assume we use aggregate BLS signatures with the standard proof-of-possession mitigation against rogue-key attacks, so that public keys and signatures are very cheaply aggregated by simply adding them together. On the same hardware above, we benchmarked the `blst` library at `https://github.com/supranational/blst`. The cost of signing or verifying one BLS signature is well under 1ms, and the cost of adding public keys and signatures in the aggregation process can be effectively ignored (at least for quorums of size up to a few hundred). To aggregate many unverified BLS signatures, a party P can very cheaply aggregate the unverified signatures and then verify the result. If the aggregate verification fails, P will have to perform a much more expensive search to find out which of the individual signatures were bad. However, once the bad signatures are found, since the parties that contributed those signatures must be corrupt, P can simply ignore all signatures (and indeed all messages) sent from these parties going forward. This works because we are assuming the signatures are sent over authenticated channels (although P cannot publicly prove their corrupt behavior, unless the BLS signatures are themselves authenticated using some cheaper digital signature, such as EdDsa). Thus, over the long run, the cost of verifying and aggregating a set of individual signatures is essentially just the cost of one BLS signature verification. Similarly, when a party P receives an aggregate signature from another party, if the verification of that aggregate signature fails, P can simply ignore that party going forward.

The other main computational cost to consider is that of hashing. On the same hardware mentioned above, the `openssl` implementation of SHA256 runs at a speed of 2GB/s.

With these benchmarks, and additional assumptions on network bandwidth and latency, we can estimate the performance (latency and throughput) of the protocol (in the optimistic setting). We shall assume network bandwidth of 1Gb/s (i.e., 125MB/s) and that the protocol is running over a WAN, so that there is essentially no contention for network bandwidth among the parties. Specifically, our assumption is that all parties can simultaneously transmit to the network at a rate of 1Gb/s. We shall assume a network latency of 100ms (so it takes 100ms for a packet to travel from P to Q once P has transmitted the packet, which is generally consistent with round-trip times reported in https://www.cloudping.co/grid/p_90/timeframe/1D).

The protocol's performance will depend on:

- *transmission delay:* the delay per slot induced by network bandwidth,
- *propagation delay:* the delay per slot induced by the network latency,
- *computation delay:* the delay induced by computation.

The optimistic consecutive-proposal latency is just the sum of these delays and throughput is the block size β divided by the sum of these delays. Here, we will assume that β is the number of *bytes* in a block. Of course, β also impacts transmission and computation delay.

We will make one small change to the protocol that will streamline its execution. Namely, instead of using an $(n, n-2t)$-erasure code, we will use an $(n-1, n-2t-1)$-erasure code, and adopt the convention that the leader does not hold a fragment. We note that with this change, the encoding of a block is still at most 3β bytes, and that the above benchmarks for $n = 97$ are still valid. With this change, the way the block data flows through the network in a given slot is as follows:

- the leader encodes a block of size β as a codeword of size $\approx 3\beta$, and transmits to each of the $n-1$ other parties its fragment, which has size $\approx 3\beta/n$, so that the leader transmits a total of $\approx 3\beta$ bytes across the network.
- each party other than the leader broadcasts its fragment of size $\approx 3\beta/n$ to the $n-2$ other parties (besides itself and the leader), so each such party transmits a total of $\approx 3\beta$ bytes across the network.

Assuming fragments are sufficiently large, each fragment can be broken up into many packets, and a simple "packet-switching pipeline" strategy can be used to minimize the transmission delay. Specifically, the leader begins by sending to each other party P the first packet of P's fragment, then it sends to each other party P the second packet of P's fragment, and so on; at the same time, when a party P receives one packet of its own fragment from the leader, it immediately broadcasts that fragment to all other parties. One sees that with this simple "packet-switching pipeline" strategy, the transmission delay per slot is roughly 3β bytes divided by the network bandwidth available to each party (without pipelining, it would be twice as much). With a network bandwidth of 1Gb/s, this translates into a transmission delay per slot of about 25ms for every 1MB of (original, unencoded) block data.

Next, consider propagation delay. This is twice the network latency, so $2 \cdot 100\text{ms} = 200\text{ms}$ under our assumptions. To make things more concrete, let us choose a block size that roughly balances transmission and propagation delay, so a block size of 8MB. With a block size this large, and for $n \approx 100$, the size of each fragment is $\approx$ 240KB, large enough to make the simple "packet-switching pipeline" strategy feasible (with packets of size $\approx 1KB$, a party can transmit one packet to each other party in time under 1ms).

Third, consider computation delay. There are several components to this:

- *erasure coding:* the leader encodes β bytes of data, and then each receiving party decodes and encodes the same amount of data; with our given estimates (for $n = 97$), this takes $2 \cdot 4\text{ms} + 16\text{ms} = 24\text{ms}$. Using multiple cores, this could likely be reduced significantly.
- *hashing:* the leader hashes 3β bytes of data, and then each receiving party hashes the same amount of data; with our given estimates, this takes $2 \cdot 12\text{ms} = 24\text{ms}$. However, the hashing done by the leader can overlap entirely with the transmission delay (the hashing can be done concurrently with the transmission of the fragments). For the receiving parties, in a typical execution, of the 3β bytes of data they need to hash, at least 2β bytes of hashing can overlap with the transmission delay (assuming the hashing is done as packets are received). If they receive support shares from all other parties, no more hashing needs to be done. In the worst case, they need to hash β bytes (after the re-encoding step), and with our given estimates, this takes 4ms. Using multiple cores, this could likely be reduced even more.
- *signing and aggregating:* each party generates a support share and then forms a support certificate. With our given estimates, this takes a total of 2ms. However, the 1ms of time spent forming a support certificate easily overlap the above 4ms of hashing time (assuming multiple cores). We do not count here the cost of processing commit shares and certificates, as these can be performed on a separate core.

This all adds up to a computation delay of $24\text{ms} + 4\text{ms} + 1\text{ms} = 29\text{ms}$, and we will round this up to 40ms to be conservative (although by exploiting multiple cores, it could be much less).

With these parameters, we estimate the total delay per slot as:

- 200ms transmission,
- 200ms propagation,
- 40ms computation.

This translates to a throughput of 8MB every 440ms, so about 18MB per second. The optimistic consecutive-proposal latency is 440ms and the optimistic proposal-commit latency is that plus about 100ms, so about 540ms.

To get a better understanding of this setting, consider the following example timeline. Suppose that at time t a leader starts transmitting the packets of a block. By time (roughly) $t + 100\text{ms}$ the other parties start echoing these packets. By time (again, roughly) $t + 200\text{ms}$ the leader finishes transmitting packets and transmits the remaining elements of its block proposal. By time $t + 300\text{ms}$ all of these packets and remaining elements have been echoed by the other parties; moreover, by this same time, the other parties have validated the block proposal and have broadcast a signature share on a corresponding support message. By time $t + 400$ms, the other parties have received all the fragments and other data they need, and then perform 40ms of computation to finish the slot with a block in the complete block tree by time $t + 440$ms.

Note that all of the above estimates are essentially independent of n. Indeed, the component of propagation and computation delay that depends on n will be a very small fraction of the total for block sizes of at least 1MB and for n up to several hundred.

Later in Section 5 we will discuss a variation of DispersedSimplex that supports "stable leaders", and in Section 6 we will discuss variations that utilize better techniques for data dissemination, and we will argue that all of these variations, both in isolation and in combination with one another, lead to even better performance.

4 Some implementation details and minor variations

4.1 Implementing the block proposal validation logic

To validate a proposal for a block B in slot v whose parent is a block B' in slot v', a party needs to check if its complaint pool contains complaint certificates for slots $v' + 1, \ldots, v - 1$. Here is a simple, practical way to do this.

Suppose that when a party enters the loop iteration for slot v, the highest slot number for which it has committed is v_{com}. We know by the Incompatibility of Complaint and Commit Property, there can never be a complaint certificate for slot v_{com}. So the party can maintain two data structures.

- A doubly linked list of those slots in the range $\{v_{\text{com}}, \ldots, v - 1\}$ for which it *does not* have a complaint certificate, in order from lowest to highest.
- A lookup table from $\{v_{\text{com}}, \ldots, v - 1\}$ to nodes in this doubly linked list — this table could just be a dynamic, circular array.

Then, the party can perform the following operations:

- Whenever a new complaint certificate appears for a slot in the range $\{v_{\text{com}}, \ldots, v - 1\}$, it accesses the corresponding node via the lookup table and removes it from the linked list.
- When the value of v_{com} or v is increased, it updates both the lookup table and linked list in the obvious way.

For each slot, a constant amount of work is performed to maintain this data structure. Moreover, at any point in time, a party can find in constant time the highest slot number $v^* < v$ for which it has complaint certificates for slots $v^* + 1, \ldots, v - 1$.

4.2 Simple variations

We mention here a few simple variations of DispersedSimplex.

- *Choice of parent block.* In the protocol, the leader in slot v proposes a new block whose parent is B_{prev}. In fact, the leader is free to choose as the parent block any block B' for a slot v' such that $v' < v$ and the leader's complaint pool contains complaint certificates for each slot $v' + 1, \ldots, v - 1$.

- *Moving on from bad blocks.* In the protocol, in managing the complete block tree, when a party reconstructs the payload and finds that it is bad (either $\perp$ or otherwise invalid), it effectively just ignores the block and the slot will eventually time out. In a variation, parties could choose to move on to the next slot right away. To do this, we also have to modify the protocol in two ways. First, each party should record the fact that is there was bad block in a given slot. Second, the logic for block proposal validation should change, so that instead of checking that we have a complaint pool contains complaint certificates for each slot $v'+1, \ldots, v-1$, we check that for each of these slots, we either saw a bad block or we have a complaint certificate.
- *Optimizing small payloads.* For small payloads, instead of erasure coding the payload and dispersing fragments, the leader could just disperse the payload directly. A support share would also contain the payload as well. Alternatively, we could use an erasure code with different parameters that was more suitable for small payloads.

5 Stable leaders

In many settings, it makes sense to keep a leader that is doing a good job in place for an extended number of slots. There are a number of advantages to this. One advantage is with respect to the most common type of failure, when a party is temporarily crashed or rebooting. In this case, whenever such a crashed party is selected as leader, the protocol has to wait sufficiently long to "time out" and move to the next slot, effectively wasting the equivalent of a few slots. In contrast, if a leader by default stays in place for, say, 1000 slots, when we come to a crashed leader, we will still waste the equivalent of a few slots, but this will be a much smaller percentage of all slots. Another advantage is that if transactions are being submitted to the system by external clients, then (just as in classical PBFT) these transactions can typically just be sent to a stable leader. Yet another advantage, as we will discuss below, is that a stable leader can drive the protocol even faster, achieving both higher throughput and lower latency.

The Simplex protocol has such a very natural internal logic to it that the logic for maintaining stable leaders suggests itself almost immediately. Let us say that by default a leader will stay in place for a certain number of consecutive slots, which we call an *epoch*. For example, one epoch might be 1000 consecutive slots.

- So that we can move to the next epoch as soon as we detect a faulty leader, we shall adopt the convention that a complaint certificate for a slot v effectively covers the rest of the epoch containing v.
- In order to maintain safety, this means that any party that issues a complaint share for a slot v must abstain from issuing a commit certificate in slot v *and all remaining slots of the interval containing* v.
- This means that once one honest party issues a complaint share for a slot v, it may not be possible to commit a block in slot v *or in any of the remaining slots of the interval containing* v, even though blocks may continue to be supported and added to the complete block tree.
- Therefore, in order to maintain liveness, we introduce logic that prevents parties from moving too far ahead of the slot of the last committed block in an epoch.

The details of our protocol, which we call **StableDispersedSimplex**, are in Fig. 2. Note that for any slot number v, $\mathsf{begin}(v)$ denotes the first slot number of the epoch containing v, while $\mathsf{end}(v)$ denotes the last slot number in an epoch. The value k is a constant parameter, which can be set to

StableDispersedSimplex: *main loop for party P_j*

```
v_last ← 0, v ← 1
repeat forever
    t_start ← clock()
    done ← proposed ← supported ← complained ← false
    if v = begin(v) then complainedInEpoch ← false   //   new epoch
    while not done do
        wait until either:
            the certificate pool contains a complaint certificate for any slot in [begin(v) . . v] ⇒
                done ← true, v ← end(v) + 1 //  go to next epoch
            the complete block tree contains a block for slot v and
              ( v = end(v) or there is a committed block for all slots in [begin(v) . . v − k] ) ⇒
                if not complainedInEpoch then broadcast a commit share for v
                done ← true, v_last ← v, v ← v + 1   //   go to next slot
            not complained and ( complainedInEpoch or clock() > t_start + Δ_timeout ) ⇒
                complained ← complainedInEpoch ← true
                broadcast a complaint share for slot v

            // The rest is the same as in Fig. 1
            leader(v) = P_j and not proposed ⇒
                proposed ← true
                generate block proposal material B, (f_1, π_1), . . . , (f_n, π_n)
                for i ∈ [n]: send BlockProp(B, f_i, π_i) to P_i
            not supported and received from leader(v) a valid block proposal BlockProp(B, f_j, π_j) ⇒
                supported ← true
                generate a signature share σ_j on Supp(B)
                broadcast the support share SuppShare(B, σ_j, f_j, π_j)
```

Fig. 2. Logic for main loop of StableDispersedSimplex protocol for party P_j

1 or any other small positive integer. The logic to go to the next slot on seeing an approved block ensures that the approved blocks do not get more than k slots ahead of the committed blocks (and if the network is well behaved and the leader is honest, it should never get more than 1 slot ahead).

The protocol makes use of the identical subprotocols for maintaining the certificate pool and complete block tree. The logic for generating block proposals is identical to that in the basic protocol.

The logic for validating block proposals is the same as in the basic protocol, except as follows.

- First, if $v > \mathsf{begin}(v)$, we require that $v = v' + 1$, which enshrines the fact that an honest leader should propose blocks with consecutive slot numbers.
- Second, instead of checking that the certificate pool contains complaint certificates for slots $v' + 1, \ldots, v - 1$, we check that it contains complaint certificates that effectively cover this interval — that is, for each $w \in [v' + 1 \mathinner{..} v - 1]$, there exists a complaint certificate for a slot u such that $w \in [u \mathinner{..} \mathsf{end}(u)]$. It is an easy exercise to generalize the data structures and algorithms in Section 4.1 to work in this setting.

One sees that this protocol is identical to the basic protocol if all epochs are of size 1.

5.1 Analysis

We sketch here the main ideas of the safety and liveness analysis for this protocol.

The basic properties in Section 3.1 hold here as well, except that the *Incompatibility of Complaint and Commit Property* generalizes here as follows: if a complaint certificate for a slot v has been produced, then it is impossible to produce a commit certificate for any slot in $[v \mathinner{..} \mathsf{end}(v)]$. This follows from the Quorum Intersection Property and the fact that if an honest party issues a complaint share in slot v, it will not issue a complaint share in v or any subsequent round in the same epoch as v.

Lemma 3.1 holds for this protocol as stated. The proof of Lemma 3.1 go through with essentially no change, other than to note the fact that we use complain certificates that cover the interval $[v' + 1 \mathinner{..} v - 1]$.

As for Lemma 3.2, the statement may be adjusted as follows:

Lemma 5.1 (Liveness I — stable leader version). *Consider a particular slot $v \geq 1$ and suppose the leader for slot v is an honest party Q. Suppose that the first honest party P to enter the loop iteration for slot v does so at time t. Further suppose that the network is δ-synchronous over the interval $[t, t + 3\delta]$ for some δ with $\Delta_{\text{timeout}} \geq 3\delta$. Then before time $t + 3\delta$, each honest party will reach a loop iteration $\geq v$. In addition, if each honest party issues a commit share in rounds $\mathsf{begin}(v), \ldots, v - 1$, then each honest party will finish loop iteration v before time $t + 3\delta$, by adding Q's proposed block B to its complete block tree and issuing a commit share for round v.*

Proof. The proof goes through with essentially no change in the case where v is the first slot in an epoch. For later slots in the epoch, we need to add the extra assumption that each honest party issued a commit share for all previous slots in the epoch — and so did not issue a complaint share in those slots. This guarantees that before time $t + \delta$ all honest parties will enter the loop iteration for slot v, and that before time $t + 2\delta$, not only will all honest parties issue support shares for Q's proposal but will also commit the block for slot $v - 1$. Therefore, before time $t + 3\delta$, each party will finish the loop iteration for slot v as stated. □

The lemma is stated as it is so that by repeated application of the lemma, it follows that so long as the network remains appropriately synchronous, an honest leader will continue to get all of its proposals committed.

Lemma 3.3 holds for this protocol essentially as stated – the conclusion would be better worded as "all honest parties have entered the loop iteration for some slot $w > v$". The proof only needs to be changed to reflect the fact that before time $t + \delta$, every honest party either enters the loop iteration for slot v or moves to the next epoch because of a complaint certificate for some round in $[\mathsf{begin}(v) \mathinner{\ldotp\ldotp} v - 1]$. In the latter case, before time $t + 2\delta$, all honest parties will have moved to the next epoch.

5.2 Improved performance through stability

As mentioned above, performance can be improved by having stable leaders. To see how, let us return to the concrete example in Section 3.5, with the parameters used there: $n \approx 100$ parties connected over a WAN, 1Gb/s bandwidth, 100ms latency, and an 8MB block size.

In the example timeline we gave there, if the leader starts transmitting the packets of a block at time t, then by time (roughly) $t + 200$ms the leader stops transmitting, but the other parties will not finish the slot until time (again, roughly) $t + 440$ms. With a constantly rotating leader, the leader for the next slot will wait until this time before it begins transmitting the packets of its block. However, a stable leader can start transmitting these packets already at time $t + 200$ms. Indeed, between time t and $t + 200$ms, it could have gathered the transactions for its next block (and even performed the erasure encoding of that block), so that it can start transmitting the these packets right away at time $t + 200$ms.

Thus, throughout an epoch where the leader is honest and the network is synchronous, we basically get another level of pipelining, with the leader starting a new slot every 200ms. In somewhat more detail:

- the leader transmits its first proposal from time t to $t + 200$ms, its second proposal from time $t + 200$ms to $t + 400$ms, and so on;
- each receiver echoes the first proposal from time $t + 100$ms to $t + 300$ms, the second proposal from time $t + 300$ms to $t + 500$ms, and so;
- each receiver downloads fragments for the first proposal from time $t + 200$ms to $t + 400$ms, for the second proposal from time $t + 400$ms to $t + 600$ms, and so;
- each receiver
  - assembles fragments for the first proposal starting at time $t + 400$ms, placing the first block in the complete block tree by time $t + 440$ms (and in time to validate the second proposal at time $t + 500$ms),
  - assembles fragments for the second proposal starting at time $t + 600$ms, placing the second block in the complete block tree by time $t + 640$ms (and in time to validate the third proposal at time $t + 700$ms),
  - and so on;
- each receiver commits the first block by time $t + 540$ms, the second block by time $t + 740$ms, and so on.

Note that in these circumstances, all parties will essentially fully utilize all available network bandwidth. Achieving all this assumes multi-threading on a few cores.

This translates to a throughput of 8MB every 200ms, so about 40MB per second. The optimistic consecutive-proposal latency is 200ms. The optimistic proposal-commit latency remains the same as in the rotating leaders version, so about 540ms.

Finally, we note that while the stable leader may nearly saturate its upload bandwidth, it is not consuming very much download bandwidth, which leaves plenty of bandwidth available for downloading transactions that are submitted directly to the stable leader by external clients.

5.3 Performance, quality, and censorship attacks

One problem with the stable leader regime is that it opens up the protocol to “performance attacks” [CWA+09,ACKL11]. In such an attack, a corrupt leader (perhaps in collusion with other parties) “slow walks” the protocol, making it perform much below its potential, but not slow enough to trigger sufficiently many complaints to dislodge the leader. For example, in StableDispersedSimplex, a corrupt leader could drive the protocol through an entire epoch at a rate of just below one block per Δ_{timeout} time units. In an implementation, the parameter Δ_{timeout} may be set to a conservatively large value, and thus such a leader can drive the protocol at a much slower pace than an honest leader would, even when the network is synchronous. Another problem is that while a corrupt leader may deliver blocks at a fast rate, those blocks could contain a small number of transactions, or a small number of “quality” transactions (according to some quality metric).

We can use the complaint mechanism of StableDispersedSimplex to mitigate against both types of attacks. Each party can monitor the throughput and quality of the stream of transactions it delivers, and issue a complaint if either of these metrics fall below some prescribed threshold for some sustained period of time. Each party will make these decisions based on local information, but as soon as more than $n - 2t$ honest parties issue complaints against a leader, no more blocks can be committed in the epoch, and so all remaining honest parties will soon complain and dislodge the leader. Because the mechanism in StableDispersedSimplex to fail over to a new leader is fairly lightweight, this approach should be quite effective in practice.

A more insidious problem with the stable leader regime is that it allows a corrupt leader to selectively censor transactions. One could, in principle, use the complaint mechanism to dislodge a censoring leader. However, building a reliable censorship detector may not be practical. In that case, a reasonable compromise may be to limit the length of epochs — most of the performance benefits of stable leaders will already be realized by using relatively short epochs that extend for, say, a hundred slots, so that a censoring leader will not be in place for a very long period of time.

6 Using better data dissemination techniques

Recently, [Loc24,LS24] presented techniques to reduce the communication complexity of reliable broadcast using erasure codes with better parameters. That is, instead of using an $(n, n-2t)$ erasure code as we do here (and in most of the literature on communication-efficient reliable broadcast, starting with [CT05]), the papers [Loc24,LS24] show how to use $(n, n-t)$ erasure codes. This essentially reduces the encoding expansion rate. For example, while an $(n, n-2t)$ erasure code leads to an encoding expansion rate of 3, using an $(n, n-t)$ erasure code leads to an expansion rate of just 1.5. For long messages of size β, using an $(n, n-2t)$ code generally leads to a communication complexity of $\approx 3n\beta$. However, using an $(n, n-2t)$ code does not immediately lead to a communication complexity of $\approx 1.5n\beta$, as one might expect — some additional communication is required to maintain the reliable broadcast security properties. The paper [Loc24] shows how to reduce communication complexity to $\approx 2n\beta$ in the worst case. The paper [Loc24] also shows how to reduce communication complexity to $\approx 1.5n\beta$ during periods of synchrony and assuming very few parties are corrupt, either crashed or actively malicious; however, the particular strategy in [Loc24] sacrifices optimistic responsiveness. The paper [LS24] shows how to reduce communication complexity to $\approx 1.5n\beta$ even in the worst case.

A naive adaptation of any of these protocols to DispersedSimplex would significantly increase the optimistic proposal-commit latency. We show here how to adapt the techniques of [Loc24] and [LS24] to DispersedSimplex without increasing this latency at all. We start by showing in some detail how to adapt the techniques of [Loc24], and then below in Section 6.3 we briefly sketch how to adapt the techniques of [LS24], which gives better communication complexity in the presence of actively malicious parties, but is somewhat more complicated.

We now describe how to adapt the techniques of [Loc24] to obtain a simple variant of DispersedSimplex with the following properties:

- When the leader is honest and the network is synchronous, its communication complexity is $\approx 1.5(n+t^*)\beta$, where $t^* < n/3$ is the number of *actively* malicious parties (not counting crashed parties), and β is the payload size.
- In the worst case, its communication complexity is $\approx 2.5n\beta$.
- It is optimistically responsive and has the same optimistic proposal-commit latency and optimistic consecutive-proposal latency as DispersedSimplex.

We call this protocol **DispersedSimplex***. Recall the communication complexity of DispersedSimplex is essentially $3n\beta$. So even in the worst case, protocol DispersedSimplex* improves the communication complexity. In the optimistic setting, when the leader is honest and the network is synchronous, the communication complexity improves further, depending on the number t^* of actual, actively malicious parties. Arguably, we expect t^* to be typically quite small in practice; perhaps more importantly, with respect to communication complexity, the performance degrades gracefully as t^* increases. We stress that all of this achieved without sacrificing optimistic latency or optimistic responsiveness.

Besides using an $(n, n-t)$ erasure code rather than an $(n, n-2t)$ erasure code, the only difference between DispersedSimplex* and DispersedSimplex is the following change to the logic of the subprotocol managing the complete block tree — no changes to the high-level protocol in Section 2.4 are needed. Whenever a party P_j successfully adds a block B to its complete block tree, it broadcasts an "advertisement" $\mathsf{BlockAdvert}(B)$, to all parties (including itself) from which it did not receive a certified fragment for B — there can be at most t such parties. If a party P_i has received $\mathsf{BlockAdvert}(B)$ from a party P_j, it will send the message $\mathsf{BackupRequest}(B)$ to P_j, provided all of the following conditions hold:

(i) it is holding a support certificate for B;
(ii) it has not already received (and echoed) its own certified fragment (f_i, π_i) for B, either directly from the leader, or in response to another "backup request" for B;
(iii) at least Δ_{grace} time has elapsed since obtaining the support certificate for B;
(iv) at most t corresponding "backup requests" for B have been sent to other parties;
(v) it has not already sent $\mathsf{BackupRequest}(B)$ to P_j.

When P_j receives such a "backup request" from P_i in response to a "advertisement" for B that it sent to P_i, party P_j will send to P_i a message that contains P_i's missing certified fragment (f_i, π_i)

(recall that P_j has computed all the certified fragments for B as part of the decoding logic). Party P_i can then echo this "backup fragment" to all other parties, who may use this "backup fragment" in combination with other certified fragments (backup or otherwise) to reconstruct B. Note that in condition (iii), Δ_{grace} is a parameter that should be set to allow an honest leader to deliver the certified fragment P_i is expecting. It is a "grace period" that ensures that in periods of synchrony when the leader is honest, P_i will not waste its own download bandwidth or any other honest party's upload bandwidth for a fragment that the leader will deliver to P_i. This parameter should be set to the nominal network delay expected during periods of synchrony, or just a little higher. Condition (iv) ensures that P_i sends at most $t+1$ "request backups", which also limits the usage of P_i's download bandwidth but ensures at least one honest party will receive its request.

The above logic ensures the following completeness property:

> *If an honest party Q adds a block B to its complete block tree, then eventually all parties will do so. Moreover, if the block is added at a time t at which the network is δ-synchronous, it will appear in every party's complete block tree before time $t + 4\delta + \Delta_{\text{grace}}$.*

To see this, observe that

- by time $t + \delta$ all honest parties will have a support certificate for B and an advertisement for B from Q,
- by time $t + \delta + \Delta_{\text{grace}}$ they will send (if necessary) backup requests for their fragment to Q,
- by time $t + 3\delta + \Delta_{\text{grace}}$ they will echo these fragments, and
- by time $t + 4\delta + \Delta_{\text{grace}}$ all honest parties will be holding $n - t$ certified fragments for B.

This property ensures that Lemmas 3.2 and 3.3 still hold, provided the time bounds are all adjusted accordingly. Indeed, one just had to add the term $3\delta + \Delta_{\text{grace}}$ to all time bounds, since this is the additional time it may take for all honest parties to enter the loop iteration for a given slot once one honest party does so. While these adjustments can increase the amount of time it takes to recover from a failed leader, it seems like it should be a worthwhile tradeoff in practice, especially in the stable leader regime (see Section 5), which one can also adapt in the obvious way to DispersedSimplex*. Let us call this protocol **StableDispersedSimplex***. One observation to note: in the stable leader regime, a smaller value of $\Delta_{\text{timeout}} \geq 4\delta$ may be used for all slots in an epoch other than the first slot (only in the first slot do we need to budget time to recover from a bad leader).

The above logic also ensures that during periods of δ-synchrony, where $\Delta_{\text{grace}} \geq \delta$, if the leader is honest, no honest party will send backup requests for that leader's block. Therefore, each honest party will transmit at most t^* backup fragments, where $t^* < n/3$ is the number of *actively* malicious parties (not counting crashed parties). This leads to the optimistic communication complexity estimate $1.5(n+t^*)\beta$ mentioned above. Even in the worst case, each party will transmit at most t backup fragments in response to a backup request, and at most t parties will broadcast a second (backup) fragment (see argument in [Loc24]), which leads to the worst case communication complexity estimate $2.5n\beta$ mentioned above.

6.1 Concrete estimates

Let us return to the concrete example in Section 3.5, with the parameters used there: $n \approx 100$ parties connected over a WAN, 1Gb/s bandwidth, 100ms latency. In this section, we will optimistically assume that the number t^* of actively malicious parties is very small, so that we can effectively ignore the bandwidth consumed by sending "backup fragments" to such parties (alternatively, we could use the construction sketched in Section 6.3, which gives better communication complexity bounds in the presence of actively malicious parties).

Instead of an 8MB block size as in Section 3.5, we will use 16MB block size. The reason for this is because the expansion rate of the code is half as much as before. Therefore, we can transmit twice as much data in the same amount of time (assuming, as we are, that t^* is small). In addition, the time it takes for encoding, decoding, and hashing 16MB blocks is about the same as for 8MB blocks For hashing, this is clear, because of the better expansion rate of the code. For the encoding and decoding times, this was verified experimentally using the `reed-solomon-simd` library.

Analogous to what we did in Section 3.5, we modify the protocol slightly, using a $(n-1, n-t-1)$ code instead of an $(n, n-t)$ code, so that the leader never holds or disperses its own fragment.

All of the performance estimates in Section 3.5 hold here as well, except that now we are processing 16MB blocks instead of 8MB blocks. Therefore, we get a throughput of 16MB every 440ms, so about 36MB per second. The optimistic consecutive-proposal and consecutive-proposal latencies are exactly the same: 440ms and 540ms, respectively.

Now consider StableDispersedSimplex*. All of the performance estimates in Section 5.2 hold here as well, except that now we are processing 16MB blocks instead of 8MB blocks. Therefore, we get a throughput of 16MB every 200ms, so about 80MB per second. The optimistic consecutive-proposal and consecutive-proposal latencies are exactly the same: 200ms and 540ms, respectively.

6.2 Performance attacks

StableDispersedSimplex* is subject to the performance attack discussed in Section 5.3, by which a corrupt leader may drive the protocol at a rate of just below one block per Δ_{timeout} time units. However, despite fact that the advertise/request mechanism to deliver backup fragments can add additional delay when the leader is corrupt, this additional delay will not exacerbate this performance attack — the leader cannot drive the protocol any slower than this for an extended period of time. Moreover, as discussed in Section 5.3, the protocol can be adapted to monitor performance and dislodge a leader if throughput drops below some target value.

6.3 Improving the communication complexity in the presence of actively malicious parties

One can also adapt the techniques of [LS24] to DispersedSimplex, which leads to an optimistic communication complexity of $1.5n\beta$ (independent of t^*) and a worst case communication complexity of $2n\beta$. However, this comes at the cost of a considerable increase in the complexity of the protocol logic, as well as increase in its computational cost, but not one that should impact overall performance in a well-engineered implementation. We sketch here the basic ideas of this technique. As we will see, this technique works best in the stable leader regime (see Section 5), so we will assume that we are working exclusively in that regime here. We call this new variant **StableDispersedSimplex†**.

The main idea, which we borrow from [LS24], is to use a two-level encoding scheme. As above, a block payload M of size β is encoded using an $(n, n-t)$ code to obtain fragments $f_1, \ldots, f_n$, each of size at most $1.5\beta/n + O(\log n)$. In addition, each fragment f_i is encoded using an $(n, n-2t)$ code to obtain "minifragments" $\phi_{i1}, \ldots, \phi_{in}$, each of size at most $4.5\beta/n^2 + O(\log n)$. A two-level Merkle tree is formed: the top-level is a tree with a root r and n leaves $r_1, \ldots, r_n$, where each r_i is the root of a Merkle tree whose leaves are the hashes of the minifragments $\phi_{i1}, \ldots, \phi_{in}$.

To disseminate a block, a leader will send the fragment f_i to each party P_i as usual, along with the Merkle path π_i from r to r_i. To validate f_i, party P_i will compute the minifragments $\phi_{i1}, \ldots, \phi_{in}$, form their Merkle tree with root r_i, and validate that against the Merkle path π_i.[2]

When decoding fragments for a given block B, a party P_j will decode $n - t$ fragments as usual, obtaining the payload M and all of its fragments. It will then encode each fragments as a vector of minifragments and build the two-level Merkle tree as above, and compare this to the Merkle tree root for B.[3] Only then will P_j add B to its complete block tree. In addition, P_j will send to each party P_i from which it did not receive a fragment the "backup minifragment" ϕ_{ij} — so instead of sending full-sized backup fragments as in DispersedSimplex*, it only sends very small backup minifragments.

Note that a party P_i that is holding a support certificate for a block B, but did not receive its fragment for B, can still reconstruct its fragment for B by collecting $n - 2t$ corresponding minifragments, and then echo this fragment to all parties. However, the completeness property we had for adding blocks to the complete block tree is now lost. Before, if a single honest party added a block B to its complete block pool, we were sure that every honest party would eventually obtain and echo its fragment for B, and so every honest party would eventually add B to its complete block tree. Now, however, we cannot make that claim when the leader is corrupt. What we can show, however, is that if $n - 2t$ parties honest parties add B to their complete block pool, then all other honest parties will eventually do so (within two network delays). As a consequence, we can be sure that if either

- a finalization certificate for B is formed, or
- a support certificate for any immediate successor of B is formed,

then every honest party will eventually add B to their complete block tree (within two network delays).

Consider a single epoch in the stable leader regime. A series of blocks $B_1, B_2, B_3, \ldots$ will be placed in the complete block trees of the honest parties. Once a support certificate for block B_v is formed, we can be sure that each honest party will eventually hold $B_1, \ldots, B_{v-1}$ in their own complete block tree. However, without modification, the protocol could get stuck. Some small set of honest parties could finish slot v with block B_v in their complete block tree, and perhaps issue a commit share for slot v, and then move onto slot $v + 1$. Because these honest parties have $B_1, \ldots, B_v$ in their complete bock tree, all other honest parties will eventually have $B_1, \ldots, B_{v-1}$ in their complete block trees, and so will eventually enter slot v. However, all of these other honest parties could get stuck in slot v. The parties stuck in slot $v + 1$ will eventually issue complaint shares for slot $v + 1$, but will not issue complaint shares for slot v — and they cannot, as this would break safety, since they may have already issued commit shares for slot v. Likewise, the parties stuck in slot v will eventually issue complaint shares for slot v, but will not issue complaint shares for slot $v + 1$.

The above problem can be solved by adding two rules to the protocol.

[2] In a practical implementation, it may be desirable to also build a Merkle tree whose leaves are the hashes are the hashes of the fragments $f_1, \ldots, f_n$, in addition to, or folded into, the two-level Merkle tree. This allows for simpler validation of fragments and potentially enables more parallel computation.

[3] If we use the variation in footnote 2, both Merkle trees will have to be checked.

Rule 1: *Whenever a party would normally issue a complaint share in slot v, it also issues a complaint share for slot $v + 1$, provided $v + 1$ lies in the same epoch as v.*

Rule 2: *Whenever a party enters a new epoch, either because it naturally moves on from the last epoch or in reaction to a complaint certificate for the last epoch, it will broadcast full-sized backup fragments for the block in slot v_{last}, provided v_{last} lies in the last epoch (and it only needs to send backup fragments to those parties from which it did not receive a fragment).*

With Rule 2, we essentially fall back to the more expensive strategy used in DispersedSimplex*, but only once per epoch. As it does happen only once per epoch, we chose not use the advertise/request paradigm for sending backup fragments, but simply send them directly. With Rule 2, we re-establish an important invariant for the protocol: if all honest parties enter a given epoch, and some honest party exits that epoch, then all honest parties will eventually do so (within two network delays).

With Rule 1, we also re-establish the following invariant for the protocol: if all honest parties enter a given epoch, then some honest party eventually exits that epoch.

To see how these two rules work together, consider again the above situation where we had some parties stuck in slot v and others in slot $v+1$. Eventually, if nothing else happens, the parties stuck in slot v will issue complaint shares for slots v and $v + 1$ and the parties stuck in slot $v + 1$ will issue complaint shares for slot $v+1$. So a complaint certificate for slot $v+1$ will eventually be formed, and the parties stuck in slot $v + 1$ will exit the epoch, broadcasting backup fragments for the block in slot v on their way out. This ensures that the parties stuck in slot v will move on to slot $v + 1$, see the complaint certificate for slot $v + 1$, and exit the epoch as well.

We leave the details of the safety and liveness analysis to the reader. However, we claim the following:

- Lemma 3.1 holds for this protocol as stated.
  Note that Rule 1 does not impact safety, as a complaint in one slot in an epoch is effectively a complaint for the remainder of the epoch.
- Lemma 5.1 holds for this protocol, provided $\Delta_{\text{timeout}} \geq 4\delta$.
  This follows from the fact when any honest party enters any slot in an honest epoch, all other parties will enter that slot at most 2δ time units later (for the first slot in any epoch, this is guaranteed by Rule 2 above, and for any other slot, it is guaranteed because the leader is honest).
- Lemma 3.3 holds for this protocol, but with a time bound of $2\Delta_{\text{timeout}}+5\delta$ in place of $\Delta_{\text{timeout}}+2\delta$.
  This time bound may occur in the scenario above where parties are stuck in two consecutive slots. The parties stuck in the first slot may have taken time 2δ to enter that slot, and some of them may nearly time out in that slot but move onto the second slot without doing so, only to time out in the second slot. Then it takes δ more time units to transmit the complaint shares to get the parties in the second slot unstuck and 2δ more time units get the remaining parties in the first slot unstuck.

6.3.1 Concrete estimates. In terms of concrete performance, we expect StableDispersedSimplex† to perform just as well as the simpler protocol StableDispersedSimplex*. Communication costs will be strictly lower, especially when more parties are actively malicious; however, the encoding, decoding, and hashing costs will be higher. Nevertheless, with a careful implementation, and especially with the use of a few more cores, it should be possible to achieve essentially the same performance metrics as in Section 6.1, but now, even in the presence of a much larger number of actively malicious parties. Note that just as in Section 6.1, we can use an $(n-1, n-t-1)$ code for the fragments, and, in addition, we can use an $(n-1, n-2t-1)$ code for the minifragments, so that the leader never holds or disperses its own fragment or minifragments.

6.3.2 Performance attacks. The same discussion in Section 6.2 applies here: StableDispersedSimplex† is no more vulnerable to performance attacks than StableDispersed-Simplex, and the protocol can be adapted to monitor performance and dislodge a leader if throughput drops below some target value.

7 A signature-free variant

It is perhaps worth pointing out that Simplex, as well as DispersedSimplex, can be implemented without any threshold signatures at all. In this implementation, the only cryptographic assumptions needed are authenticated communication links and collision resistant hash functions (used for block chaining in both Simplex and DispersedSimplex and for Merkle trees in DispersedSimplex). The price to pay, however, is some extra latency. In this section, we sketch how this may be done in a fairly simple and modular fashion.

The basic idea is to use the "echo/ready" logic of Bracha's reliable broadcast protocol [Bra87] in place of threshold signatures. The general idea is this:

- To *issue a signature share on a message* m, a party broadcasts the object $\mathsf{Echo}(m)$.
- Whenever a party receives the same object $\mathsf{Echo}(m)$ from $n-t$ distinct parties, or the same object $\mathsf{Ready}(m)$ from $t+1$ distinct parties, he broadcasts the object $\mathsf{Ready}(m)$ (if he has not done so already).
- Whenever a party receives the same object $\mathsf{Ready}(m)$ from $n-t$ distinct parties, he *reports out a signature certificate on* m.

The analogs of the *Quorum Size Property* and *Quorum Intersection Property* hold here as well:

Quorum Size Property: If some honest reports out a signature certificate on a message m, then $n-t-t'$ honest parties must have issued signature shares on m, where $t' \leq t$ is the number of corrupt parties.

Quorum Intersection Property: If some honest party reports out a signature certificate on m and some honest party reports out a signature certificate on $m' \neq m$, then at least one honest party must have issued signature shares on both m and m'.

Another property enjoyed by this logic is a *completeness* property, which says that if one honest party reports out a signature certificate on a message m at time t, then eventually all honest parties will do so (and before time 2δ if the network is δ-synchronous over $[t, t+2\delta]$).

The above logic can be incorporated into the management of the support, commit, and complaint pools in Section 2.3.1. These pools keep track of those which certificates have been reported out, and this information is used to manage the approved block pool as in Section 2.3.2, to commit blocks as in Section 2.3.3, and to implement the logic of the main protocol as in Section 2.4. This all works because the only thing that a party in the original protocol did with a certificate was to (i) keep track of which certificates it had obtained, and (ii) ensure that all other parties obtained the same certificates.

7.1 Safety and liveness

In terms of the analysis of the safety and liveness properties of the resulting protocol, the only changes are as follows:

- In the completeness property in Section 3.1, the statement regarding the timing of the delivery of object X should read as follows: *if X appears in a party's pool time t and the network is δ-synchronous over $[t, t+2\delta]$, then X it will appear in every party's pool before time $t+2\delta$.*
- Lemma 3.1 holds without change.
- Lemmas 3.2 and 3.3 hold if we replace the assumption that the network is δ-synchronous by the assumption that it is $(\delta/2)$-synchronous.

7.2 Communication complexity

The communication complexity is the same as reported in Section 3.4.1, but with $\kappa := 1$.

7.3 Latency

If we consider the latency metrics discussed in Section 3.4.2, then all of the latency bounds essentially double. In particular, if the network is δ-synchronous, then for DispersedSimplex, as well as for Simplex, the optimistic proposal-commit latency is 6δ and the optimistic consecutive-proposal latency is 4δ. These estimates take into account the fact that if an honest leader makes a proposal at time t, then all parties will receive the proposal by time $t+\delta$; however, the other parties may need to wait until time $t+2\delta$ to obtain the data needed to validate the proposal (the support or complaint certificate for the previous slot) as they wait for Bracha's "ready amplification" logic to run its course.

As discussed in Section 3.4.2, the optimistic consecutive-proposal latency upper bounds the *optimistic reciprocal block throughput*, the reciprocal of the rate at which blocks are proposed (and committed) in a steady state where all leaders are honest and the network is appropriately synchronous. In this setting, the optimistic reciprocal block throughput is actually bounded by $\approx 3\delta$. To see this, let us define p_v to be the time at which the slot-v leader proposes a block B_v, and s_v to be the time at which the following event happens: either

- all honest parties have issued a support share for B_v, or
- some honest party approves B_v.

With this definition, and by Bracha's "echo/ready" logic, we see that by time $s_v + 2\delta$, all honest parties will have approved B_v. We have

$$s_{v+1} \leq s_v + 3\delta. \tag{1}$$

To see this, note that by time $s_v + 2\delta$ all honest parties, including the slot-$(v+1)$ leader, will have approved B_v, and so by time $s_v + 3\delta$ all honest parties have either issued a support share for B_{v+1} or approved B_{v+1}. We also clearly have

$$p_v \leq s_v. \tag{2}$$

In addition, we have

$$s_v \leq p_v + 2\delta, \tag{3}$$

which again follows from Bracha's "echo/ready" logic. Therefore, if the leaders in slots $v, v+1, \ldots, v+k$ are all honest, we have

$$\begin{aligned} p_{v+k} &\leq s_{v+k} \quad \text{(from (2))} \\ &= p_v - p_v + s_{v+k} \\ &\leq p_v - (s_v - 2\delta) + s_{v+k} \quad \text{(from (3))} \\ &= p_v + 2\delta + (s_{v+k} - s_v) \\ &\leq p_v + 2\delta + 3k\delta \quad \text{(from (1))}, \end{aligned}$$

which implies (for large k) that the optimistic reciprocal block throughput is bounded by $\approx 3\delta$.

7.4 Stable leaders

The above technique can clearly be used to make the StableDispersedSimplex protocol in Section 5 signature free as well. In this setting, the optimistic reciprocal block throughput can be reduced from $\approx 3\delta$ to $\approx 2\delta$. Moreover, the very fact that the leader is stable makes the notion of optimistic reciprocal block throughput more relevant. The main idea is the observation that a stable leader need not wait until it has reported out a support certificate for one block before proposing the next block (in contrast, if the leader is constantly rotating, the next leader must wait to report out either a timeout or support certificate for the previous slot, in order to maintain liveness). Instead, we can impose the following *pipelined proposal rule*: the leader will propose block B_{v+1} in slot $v+1$ upon

- receiving *support-echo* objects for B_v from $n-t$ distinct parties, or
- receiving *support-ready* objects for B_v from $t+1$ distinct parties.

This proposal rule ensures that at the time the leader proposes B_{v+1}, at least $n-t-t'$ honest parties have issued support shares for B_v (and hence have already approved B_{v-1}). Thus, the leader does not run too far ahead of the other honest parties.

With the values p_v and s_v defined as above, the inequalities (2) and (3) hold just as before. Based on the pipelined proposal rule, it is not hard to see that $p_{v+1} \leq s_v + \delta$. From this, we see that

$$s_{v+1} \leq \max\{s_v + 2\delta, p_{v+1} + \delta\} \leq s_v + 2\delta.$$

Therefore, by reasoning similar to that used above, we see that

$$p_{v+k} \leq p_v + 2(k+1)\delta,$$

which implies (for large k) that the optimistic reciprocal block throughput is bounded by $\approx 2\delta$.

Note also that with this pipelining, if one also takes into account transmission and computational delays, this version of the protocol should be able to sustain exactly the same level of throughput discussed in Section 5.2 (under the same assumptions).

7.5 Using better data dissemination techniques

One can easily adapt the techniques in Section 6 to work with this erasure free variant. We leave the details to the reader.

8 Comparison to other protocols

8.1 Simplex

As already mentioned above in Section 3.4.2, the optimistic proposal-commit latency (3δ) and the optimistic consecutive-proposal latency (2δ) of DispersedSimplex are the same as for Simplex. A proper comparison of the communication complexity of DispersedSimplex and Simplex is not really possible. This is because description of Simplex in [CP23] is a bit problematic: taking the description of the protocol in Section 2.1 of [CP23] literally, the size of the message in slot v is actually proportional to v itself, but elsewhere (in particular in Section 3.4 of [CP23]) it is suggested that messages are much smaller (but without any details). DispersedSimplex is optimistically responsive, just like Simplex.

8.2 HotStuff and HotStuff-2

We may also compare DispersedSimplex to HotStuff [YMR+18] and the recently proposed improvement HotStuff-2 [MN23].

8.2.1 Latency. HotStuff-2 has an optimistic proposal-commit latency of 5δ while HotStuff has a an optimistic proposal-commit latency of 7δ. Pipelined versions of these protocols can achieve an optimistic consecutive-proposal latency 2δ. Thus, (pipelined versions of) HotStuff and HotStuff-2 have the same optimistic consecutive-proposal latency of DispersedSimplex, but have worse optimistic proposal-commit latency (which is just 3δ for DispersedSimplex).

We note that HotStuff and HotStuff-2 are optimistically responsive, just like DispersedSimplex and Simplex.

8.2.2 Communication complexity. The reported communication complexity of HotStuff and HotStuff-2 is

$$O(n(\beta + \kappa + \lambda)).$$

Recall that β bounds the block size, κ the signature share/certificate size, and λ the hash size. For small blocks, specifically if $\beta \ll n(\kappa + \lambda \log n)$, this communication complexity is better than that of DispersedSimplex, which is $O(n\beta + n^2(\kappa + \lambda \log n))$, as we discussed above in Section 3.4.1. However, this reported communication cost does not actually take into account the cost of *reliable* block dissemination. In these protocols, the leader is (apparently) supposed to simply send its proposed block to each party — at least, that is what is written in [YMR+18].

This creates two problems. First, there is no mechanism specified that ensures that all honest parties obtain the payloads of committed blocks. Naive mechanisms in which parties simply poll other parties for missing blocks can easily degenerate into $O(n^2\beta)$ communication complexity: all corrupt parties could simply ask for a block from all honest parties. If information dispersal techniques are used to ensure data availability, this would again make the communication complexity quadratic in n. So at best, the communication complexity of these protocols is better only for small blocks and only assuming corrupt parties do not misbehave too much.

Second, if the description in [YMR+18] is taken literally, the communication load in HotStuff (and apparently HotStuff-2) is very *unbalanced*. This can create a communication bottleneck at the leader. Indeed, as demonstrated empirically in [MXC+16,SDPV19], it seems that for systems with moderate network size (n up to a hundred or so) and large block sizes, taking care to disseminate blocks to all parties in a way that does not create a bottleneck at the leader is more important in practice than worrying about the quadratic dependence on n in the communication complexity. In contrast, as mentioned above in Section 3.4.1, the communication load of DispersedSimplex is *balanced*. That is, each party, *including the leader*, transmits roughly the same about of data over the network. Thus, while in HotStuff (and HotStuff-2), the leader has to transmit $O(n\beta)$ bytes across the network, in DispersedSimplex, the leader (and every party) transmits $O(\beta)$ bytes across the network.

8.2.3 Concrete estimates. It would be interesting to perform a careful empirical investigation to compare the real-world performance of DispersedSimplex and (pipelined) HotStuff/HotStuff-2 under various parameter settings. However, we can attempt to make a "back of the envelope" calculation, similar to what we did in Section 3.5. With the parameters we used there (1Gb/s network bandwidth and 100ms network latency), the propagation delay per slot would be the same, so about 200ms, and the computation delay would be less. As for the transmission delay, if the block size is β bytes, then in each slot the leader has to transmit a total of $n\beta$ bytes across the network. As a specific example, let us say $n \approx 100$, so the transmission delay would be about 800ms for every 1MB of block data. This is obviously much worse than the 25ms per 1MB of block data for DispersedSimplex. With these estimates, the best possible throughput that could be achieved is 1.25MB of block data per second. More concretely, suppose we set the block size to 1MB. So ignoring computation delay (which is just a few ms),

- the throughput is about 1MB per second (vs 18MB per second for DispersedSimplex, or 80MB per second for the stable-leader version of DispersedSimplex, as discussed in Section 6.1),
- the optimistic consecutive-proposal latency is 1s (vs 440ms for DispersedSimplex, or 200ms for any of the stable-leader versions of DispersedSimplex discussed here), and
- (for HotStuff-2) the optimistic proposal-commit latency is that plus about 300ms, so about 1.3s (vs 540ms for DispersedSimplex).

In the above calculations, we saw that for an unbalanced protocol like HotStuff (or PBFT), as n increases, the throughput should decrease, and the latency should increase, while in a balanced protocol like DispersedSimplex, throughput and latency should not depend very much on n. This type of behavior has been confirmed experimentally in papers such as [MXC+16,SDPV19], although not for the exact protocols considered here. Also, while we focused on throughput and latency, there are other costs to consider — namely, the monetary (or other) costs associated with transmitting a certain *amount* of data. These costs are directly proportional to the overall communication complexity, and it is indeed true that erasure coding does inflate these costs by a factor of 3 (although this can be reduced to a factor of 1.5 using the techniques in Section 6). Another factor to potentially consider is the fact that for a balanced protocol like DispersedSimplex, the rate at which each party is transmitting is fairy constant, while for protocols like HotStuff, it is very bursty.

8.2.4 A tension between timeouts. Another issue with HotStuff-2 is that in addition to a timeout analogous to the value Δ_{timeout} used in DispersedSimplex, there is a waiting period Δ_{wait} used by the leader in some situation to ensure that is becomes aware of any "hidden locks" held by other parties that would prevent its proposal from being accepted (and thus lose the liveness property). Now, it is a well-established technique that a system might choose an initial timeout Δ_{timeout}-value, but parties might adjust this value upwards if progress is not being made for a while (which would deal with situations where the network becomes significantly slower than the design parameter for an extended period). One could obviously implement such a technique in both DispersedSimplex and HotStuff-2. Note that parties make these decisions locally and may end up with (very) different values of Δ_{timeout}. However, to preserve liveness in HotStuff-2, the leader would have to adjust Δ_{wait} as well as Δ_{timeout}. Unfortunately, if the leader's value of Δ_{wait} becomes too large relative the timeout Δ_{timeout}-values of the other parties, the other parties will time out before the leader makes its proposal. It is not clear if this is a significant problem in practice, but it is worth pointing it out as a potential problem. In contrast, in a protocol such as DispersedSimplex, if some parties have a Δ_{timeout}-value that is "too large", this will not impact liveness — it will only impact what we called above the "optimistic k-gap proposal latency", that is, the latency between proposals made between consecutive honest leaders separated by k corrupt leaders.

8.3 ICC

The Simplex protocol bears a passing resemblance to the ICC protocols ICC in [CDH+21]. The main difference is that for the ICC protocols, if the leader for a slot v is perceived to fail, then instead of simply timing out, a (somewhat complicated) fail-over mechanism is triggered that will *eventually* add a block to the complete block tree for slot v that is proposed by a different party. Latency and communication costs in the optimistic setting for protocols ICC0 and ICC1 in [CDH+21] are very similar to that of Simplex. We note that protocol ICC2 in [CDH+21] employs information dispersal techniques to get better communication complexity, but at the expense extra latency. Thus, DispersedSimplex is both simpler and more efficient than that any of the ICC protocols.

8.4 DAG-based atomic broadcast protocols

Recently, there has been a flurry of papers on DAG-based atomic broadcast protocols — for example [KKNS21,DKSS22,SGSK22,SAGL23]. One of the attractions of these protocols is that, by design, they are leaderless and thereby avoid the bandwidth bottleneck that some leader-based protocols can exhibit. Indeed, as stated in [DKSS22]:

> *decoupling transaction dissemination from the critical path of consensus is the key to blockchain scalability.*

As mentioned above, the papers [MXC+16,SDPV19] already demonstrated the importance of taking care to disseminate blocks to all parties in a way that does not create such a bottleneck. We also mentioned above that protocol ICC2 in [CDH+21] shows how to do this in a leader-based protocol, and we have shown in this paper how DispersedSimplex achieves this in a leader-based protocol with optimal proposal-commit latency. As shown in Section 5, a stable-leader variant of DispersedSimplex can achieve even better performance, and specifically, when the leader is honest and the network is synchronous, all parties will essentially *fully utilize all available network bandwidth*. Thus, it is not entirely clear to us that the leader-bottleneck problem exhibited by some earlier leader-based protocols is a valid reason to abandon leader-based protocols entirely, especially since leader-based protocols (such as DispersedSimplex) still exhibit superior (and essentially optimal) latency characteristics. Moreover, it is also not entirely clear to us that "decoupling transaction dissemination from the critical path of consensus" is an inherently good idea: while such a decoupling may be good from a software engineering point of view, as we demonstrate with DispersedSimplex, it is precisely by *tightly coupling* dissemination with consensus that *we can fully utilize network bandwidth without sacrificing optimal latency*, using a quite simple and elegant protocol.

There are many metrics on which consensus protocols may be compared. While DAG-based consensus protocols may well be superior on some metrics, it does not appear (based on our analysis) that the core metrics of common-case throughput and latency are among them.

Acknowledgments

Thanks to Benjamin Chan and Rafael Pass for helpful discussions on the Simplex protocol. Thanks to Ed Felten for suggesting the "packet-switching pipeline" strategy in Section 3.5.

References

ACKL11. Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Secur. Comput.*, 8(4):564–577, 2011.

ADVZ21. N. Alhaddad, S. Duan, M. Varia, and H. Zhang. Succinct erasure coding proof systems. Cryptology ePrint Archive, Paper 2021/1500, 2021. https://eprint.iacr.org/2021/1500.

ANRX21. I. Abraham, K. Nayak, L. Ren, and Z. Xiang. Good-case latency of byzantine broadcast: A complete categorization, 2021. arXiv:2102.07240, http://arxiv.org/abs/2102.07240.

BDN18. D. Boneh, M. Drijvers, and G. Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Paper 2018/483, 2018. https://eprint.iacr.org/2018/483.

BLS01. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In C. Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

Bol03. A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003.

Bra87. G. Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

CDH+21. J. Camenisch, M. Drijvers, T. Hanke, Y.-A. Pignolet, V. Shoup, and D. Williams. Internet computer consensus. Cryptology ePrint Archive, Report 2021/632, 2021. https://ia.cr/2021/632.

CL99. M. Castro and B. Liskov. Practical byzantine fault tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL https://dl.acm.org/citation.cfm?id=296824.

CP23. B. Y. Chan and R. Pass. Simplex consensus: A simple and fast consensus protocol. Cryptology ePrint Archive, Paper 2023/463, 2023. https://eprint.iacr.org/2023/463.

CT05. C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In P. Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005.

CWA+09. A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In J. Rexford and E. G. Sirer, editors, *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 153–168. USENIX Association, 2009. URL http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf.

DKSS22. G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In Y. Bromberg, A. Kermarrec, and C. Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022.

DLS88. C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

DW20. S. Dolev and Z. Wang. SodsBC: Stream of distributed secrets for quantum-safe blockchain. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 247–256, Los Alamitos, CA, USA, 2020. IEEE Computer Society.

KKNS21. I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. All you need is DAG. In A. Miller, K. Censor-Hillel, and J. H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021.

LAHC16. S. Lin, T. Y. Al-Naffouri, Y. S. Han, and W. Chung. Novel polynomial basis with fast Fourier transform and its application to Reed-Solomon erasure codes. *IEEE Trans. Inf. Theory*, 62(11):6284–6299, 2016.

LC12. S. Lin and W. Chung. An efficient (n, k) information dispersal algorithm for high code rate system over Fermat fields. *IEEE Commun. Lett.*, 16(12):2036–2039, 2012.

LLTW20. Y. Lu, Z. Lu, Q. Tang, and G. Wang. Dumbo-MVBA: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In Y. Emek and C. Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020.

Loc24. T. Locher. Byzantine reliable broadcast with low communication and time complexity, 2024. arXiv:2404.08070, http://arxiv.org/abs/2404.08070.

LS24. T. Locher and V. Shoup. MiniCast: Minimizing the communication complexity of reliable broadcast. Cryptology ePrint Archive, Paper 2024/571, 2024. https://eprint.iacr.org/2024/571.

MN23. D. Malkhi and K. Nayak. Extended abstract: HotStuff-2: Optimal two-phase responsive BFT. Cryptology ePrint Archive, Paper 2023/397, 2023. https://eprint.iacr.org/2023/397.

MXC+16. A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

SAGL23. A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li. Shoal: Improving dag-bft latency and robustness, 2023. arXiv:2306.03058, http://arxiv.org/abs/2306.03058.

Sch90. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

SDPV19. C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolić. Mir-BFT: High-throughput robust bft for decentralized networks, 2019. arXiv:1906.05552, http://arxiv.org/abs/1906.05552.

SGSK22. A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In H. Yin, A. Stavrou, C. Cremers, and E. Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022.

YMR+18. M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus in the lens of blockchain, 2018. arXiv:1803.05069, http://arxiv.org/abs/1803.05069.

YPA+21. L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse. DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks, 2021. arXiv:2110.04371, http://arxiv.org/abs/2110.04371.