DispersedSimplex: simple and efficient atomic broadcast

Victor Shoup

Offchain Labs
victor@shoup.net

December 18, 2023

Abstract. In this brief note, 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, while maintaining the simplicity and excellent latency characteristics of the protocol.

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 \( n \) 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 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 brief note, 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 chained together by cryptographic hashes, rooted at a special genesis block. 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 brief note, we hope to add to the Simplex story in a small way, mainly by showing how to improve its communication complexity. First, we flesh out some missing (but crucial) details of the Simplex protocol that are needed to get a protocol with acceptable communication complexity. Second, and more importantly, we modify the protocol so that leaders disperse blocks in a more communication-efficient fashion, while maintaining the simplicity and excellent latency characteristics of the protocol. We call this variation on the Simplex protocol DispersedSimplex. We give a detailed analysis of DispersedSimplex (safety, liveness, and performance), and compare it to Simplex, HotStuff, and HotStuff-2 (as well as the ICC protocols in [CDH+21]). Our presentation is fairly self contained — certainly, no knowledge of the Simplex protocol itself is assumed, but some familiarity with similar protocols (like PBFT or HotStuff) would be helpful.

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. 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. Our main insight (such as it is) is 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 \( n \) 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 \( \delta \)-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 \( \delta \)-synchronous at time \( t \).
2.1.1 Signatures. We make use of an \((n-t)\)-out-of-\(n\) threshold signature scheme. 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 \([\text{BLS}01]\) as in \([\text{BDN}18]\)) or as a threshold version of an ordinary signature scheme (such as one again based on BLS signatures as in \([\text{Bol}03]\)). 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 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 of these messages.

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 \([\text{CT}05]\)). In particular, 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 \(m\). Using a standard Reed-Solomon code, this leads to a data expansion rate of (at most) roughly 3; that is, \(\sum_j |f_i| \approx \frac{n}{n-2t} \cdot |M| < 3|M|\).

2.2 Protocol data objects

2.2.1 Blocks. A block \(B\) is of the form \(\text{Block}(v,h,r)\), where

- \(v = 0,1,\ldots\) is the slot number associated with the block (and we say \(B\) is a block for slot \(v\)),
- \(h\) is the hash of the \(B\)'s parent block, and
- \(r\) is the root of a Merkle tree for the erasure-code fragments \(f_1,\ldots,f_n\) encoding \(B\)'s payload \(M\).

2.2.2 Support, commit, and time-out shares and certificates. A support share on a block \(B = \text{Block}(v,h,r)\) from party \(P_j\) is an object of the form \(\text{SuppShare}(B,\sigma_j,f_j,\pi_j)\), where \(\sigma_j\) is a valid signature share from \(P_j\) on \(\text{Supp}(B)\), and \(\pi_j\) is a correct validation path from the root \(r\) to the leaf \(f_j\) at position \(j\). A support certificate on \(B\) is an object of the form \(\text{SuppCert}(B,\sigma)\), where \(\sigma\) is a valid signature certificate on \(\text{Supp}(B)\).

A commit share on slot \(v\) from party \(P_j\) is an object of the form \(\text{CommitShare}(v,\sigma_j)\), where \(\sigma_j\) is a valid signature share from \(P_j\) on \(\text{Commit}(v)\). A commit certificate on \(v\) is an object of the form \(\text{CommitCert}(v,\sigma)\), where \(\sigma\) is a valid signature certificate on \(\text{Commit}(v)\).

A time-out share on slot \(v\) from \(P_j\) is an object of the form \(\text{TimeOutShare}(v,\sigma_j)\), where \(\sigma_j\) is a valid signature share from \(P_j\) on \(\text{TimeOut}(v)\). A time-out certificate on \(v\) is an object of the form \(\text{TimeOutCert}(v,\sigma)\), where \(\sigma\) is a valid signature certificate on \(\text{TimeOut}(v)\).
2.2.3 Payload reconstruction. Note that while a quorum of \( n - t \) support shares for a block \( B \) is required to construct a corresponding support certificate, we can reconstruct the payload of \( B \) from a quorum of just \( n - 2t \) support shares, as follows. Using the fragments in these \( n - 2t \) support shares, we reconstruct a tentative payload \( M' \) from the fragments, compute fragments \((f'_1, \ldots, f'_n)\), and compute the root \( r' \) of a Merkle tree for \((f'_1, \ldots, f'_n)\). If \( r' = r \), then the effective payload of \( B \) is defined to be \( M' \), and otherwise, the effective payload is defined to be \( \bot \). Under collision resistance for the hash function used for the Merkle trees, any \( n - 2t \) valid support shares for \( B \) will yield the same effective payload — moreover, if \( B \) was constructed properly from a payload \( M \), the effective payload will be \( M \) (and therefore, an effective payload of \( \bot \) indicates that the party that constructed the block \( B \) was malicious). This observation is the basis for the protocols in [DW20, LTW20, 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).

2.3 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 Support, commit, and time-out pools. Each party maintains a support pool, a commit pool, and a time-out pool. Whenever a party receives a quorum of \( n - t \) support, commit, or time-out shares, and it does not already have a corresponding certificate, it will generate a certificate, add it to the corresponding pool, and broadcast the certificate to all parties. Similarly, whenever a party receives a support, commit, or time-out certificate, and it does not already have a corresponding certificate, it will add it to the corresponding pool, and broadcast the certificate to all parties.

2.3.2 Approved block pool. Each party also maintains an approved block pool. The approved block pool always contains a tree of blocks, rooted at a special genesis block \( B_{\text{gen}} := (0, *, *) \). Initially, the approved block pool only contains the genesis block. A block \( B = (v, h, r) \) is added to the pool if

- the approved block pool contains a parent block \( B' = (v', \cdot, \cdot) \) with \( v' < v \) and \( h = \text{Hash}(B') \);
- the support 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 described above (note that we may have \( M = \bot \));
- \( M \neq \bot \) and satisfies some correctness predicate that may depend on the path of blocks (and their payloads) from genesis to block \( B' \).

Unlike the support, commit, and time-out pools, when a party adds a block to its approved block pool, it does not broadcast anything to other parties. We say a block is approved by \( P \) if it belongs to the approved block pool of \( P \).
2.3.3 Block commitment. We say that a block $B$ for slot $v$ is *explicitly committed by $P$* if it is approved by $P$ and the commit pool of $P$ contains a commit certificate for slot $v$. In this case, we say that all of the predecessors of block $B$ are *implicitly committed by $P$*. The genesis block is always considered to be a committed block.

2.4 The main protocol

The logic of the main protocol $\Pi_{\text{DispersedSimplex}}$ is described in Fig. 1. In the description, $\text{leader}(v)$ 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 support, commit, time-out, and approved block pools (which are maintained as described in Section 2.3) and the objects it has received from other parties over authenticated channels.

![Fig. 1. Logic for main loop of DispersedSimplex protocol](image)

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:
compute $h := \text{Hash}(B_{\text{last}})$;
build a payload $M$ that validly extends the path in the block tree ending at $B_{\text{last}}$;
compute the erasure code fragments $(f_1, \ldots, f_n)$ of $M$;
compute the Merkle tree for $(f_1, \ldots, f_n)$ with root $r$ and validation paths $\pi_1, \ldots, \pi_n$;
Set $B := \text{Block}(v, h, r)$.

2.4.2 Validating block proposals. To check if $\text{BlockProp}(B, f_j, \pi_j)$ 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 $\text{Block}(v, h, r)$, where $h = \text{Hash}(B')$ for some (unique) approved block $B' = \text{Block}(v', \cdot, \cdot)$ in the approved block pool;
- $v' < v$;
- the time-out pool contains time-out certificates for slots $v' + 1, \ldots, v - 1$;
- $\pi_j$ is a correct Merkle validation path from the root $r$ to the leaf $f_j$ 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 an approved parent block or time-out certificate, these conditions will need to be rechecked whenever a new block is added to the approved block pool or a new time-out certificate is added to the time-out pool. We will discuss below (in Section 5.1) how to efficiently implement the test that the time-out pool contains the necessary time-out 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 chain blocks together, 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 approved by a party. Then no other block for slot $v$ can be approved by 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 any pool (support, commit, time-out, approved block) then $X$ (or its equivalent) will eventually appear in the corresponding pool of every other party. Moreover, if $X$ appears in a party’s pool at a time $t$ at which the network is $\delta$-synchronous, it will appear in every party’s pool before time $t + \delta$.

Note that the “or equivalent” qualification is necessary to account for signature certificates, if these are not necessarily unique.
For the support, commit, and time-out pools, this is clear. For the approved block pool, we are relying on the Quorum Size Property: when a support certificate for $B$ 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 Time-out and Commit Property:** It is impossible to produce both a time-out 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 time-out 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 approved by some party $Q$. Then $B$ is an ancestor of $C$ in $Q$’s approved block pool.

**Proof.** By the Incompatibility of Time-out and Commit Property, no time-out 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 approved block pool, 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 time-out certificate for slot $v$, and an honest party will issue a support share for $C$ only if it has time-out 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 $C$. \hfill \Box

### 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 $\delta$-synchronous over the interval $[t, t + 3\delta]$ for some $\delta \leq \Delta/3$. Then all honest parties will finish the loop iteration before time $t + 3\delta$ by validating $Q$’s proposed block $B$, and will eventually commit $B$. Moreover, if the network is $\delta$-synchronous over the interval $[t, t + 4\delta]$, then all honest parties will commit the block $B$ before time $t + 4\delta$.

**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 time-out certificate for slot $v - 1$ or a approved block for slot $v - 1$. 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 time-out 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 $B'$ and these time-out certificates in their own pools, 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 approved $B$. By the assumption that $\delta \leq \Delta/3$, when each honest party has approved $B$, the
time-out 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 \( \delta \)-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 \( \delta \)-synchronous over an interval \([t, t + \Delta + 2\delta]\), for an arbitrary value of \( \delta \), 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 \( t + \Delta + 2\delta \), 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 \( t + \delta + \Delta \), every honest party will have either approved a block or broadcast a time-out share for slot \( v \). In either case, less than \( \delta \) time units later all honest parties will have either approved a block or obtained a time-out certificate for slot \( v \), and hence 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

\[
O(n\beta + n^2(\kappa + \lambda \log n)),
\]

where

- \( \beta \) is a bound on the size of a block,
- \( \kappa \) is a bound on the size of a threshold signature share or certificate,
- and \( \lambda \) is a bound on the size of the hash function outputs used for Merkle trees and block chaining.

Indeed, the cost breaks down as follows:

- \( O(n\beta) \) 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,
- \( O(n^2\lambda) \) for disseminating block hashes.

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 amount of data over the network.
3.4.2 Latency. We may also measure various notions of latency. Assume the network is $\delta$-synchronous for $\delta \leq \Delta/3$. We define:

- **optimistic proposal-commit latency**: assuming the leader is honest, the time it takes for the leader’s proposal to be committed by all honest parties;
- **optimistic consecutive proposal latency**: assuming two consecutive leaders are honest, the amount of time that elapses between when they make their respective proposals.

If a given transaction is submitted to the system, 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.

For DispersedSimplex, just as for Simplex, we readily see that the optimistic proposal-commit latency is $3\delta$ and the optimistic consecutive proposal latency is $2\delta$.

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 $2\delta + k \cdot (\Delta + \delta)$.

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.

3.4.3 Other costs and concrete estimates. The computational cost of erasure coding 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](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.

Let us look at the other computational costs that may impact the performance of the protocol, starting with 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](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.

Another cost to consider is the computational cost of hashing. On the same hardware mentioned above, the openssl implementation of SHA256 runs at a speed of 2GB/s. Both the leader and the receiving parties do a significant amount of hashing. Note that when processing support shares received from other parties in a given slot, a party \( P \) may defer hashing the embedded fragments until a support certificate is obtained. In this way, \( P \) hashes just enough fragments as required by the decoding algorithm (and then hashes the additional fragments produced by re-encoding).

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 1Gbps 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 1Gbps. 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).

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 \( \beta \) divided by the sum of these delays. Here, we will assume that \( \beta \) is the number of bytes in a block. Of course, \( \beta \) also impacts transmission and computation delay.

First, consider transmission delay. In a given slot, in the “proposal” step of the protocol, the leader encodes of block of size \( \beta \) with an erasure code that expands it to a codeword of size \( 3\beta \), and transmits \( 3\beta \) bytes across the network. In the “support” step the protocol, each party transmits \( 3\beta \) bytes across the network. This means that the total transmission delay is per slot is \( 2 \cdot 3\beta = 6\beta \) bytes divided by the network bandwidth available to each party. With a network bandwidth of 1Gbps, this translates into a transmission delay per slot of about 50ms for every 1MB of (original, unencoded) block data.

Second, consider propagation delay. This is twice the network latency, so \( 2 \cdot 100\text{ms} = 200\text{ms} \). To make things more concrete, let us choose a block size that balances transmission and propagation delay, so a block size of 4MB.

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

- **erasure coding**: the leader encodes \( \beta \) 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 \( 2 \cdot 2\text{ms} + 8\text{ms} = 12\text{ms} \). Using multiple cores, this could likely be reduced significantly.
- **hashing**: the leader hashes \( 3\beta \) bytes of data, and then each receiving party hashes the same amount of data; with our given estimates, this takes \( 2 \cdot 6\text{ms} = 12\text{ms} \). However, hashing done by the leader can entirely overlap the transmission of the fragments, and the first \( \beta \) bytes of hashing done by each receiving party can overlap entirely with the decoding (assuming multiple
cores). Altogether, this would reduce the delay caused by hashing by 8ms. Using multiple cores, this could likely be reduced even more.

- **signing and verifying**: each party generates a support share and then forms a support certificate. Also, during a given slot, the parties will be doing the same for a commit certificate from the previous slot. With our given estimates, this takes a total of 4ms. However, most, if not all, of these computations can easily overlap with other delays such as hashing, decoding, and transmission (assuming multiple cores and that data is transmitted in a convenient order).

This all adds up to a computation delay of 28ms, which we will round up to 30ms. We noted that by taking advantage of opportunities for parallelism, the net computation delay could be as low as 16ms (or even lower). We also note that the erasure code speed can vary a bit depending on the exact values of \( n \) and \( t \), and as \( n \) ranges over values up to 100, this speed can a bit faster or slower than the time we reported for \( n = 97 \). Nevertheless, taken all together (opportunities for parallelism, variations in erasure code speed), for all \( n \) up to around 100, the estimate of 30ms is a reasonable upper bound on the computation delay.

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

- 200ms transmission,
- 200ms propagation,
- 30ms computation.

This translates to a throughput of 4MB every 430ms, so about 9.3MB per second. The optimistic consecutive proposal latency is 430ms and the optimistic proposal-commit latency is that plus about 100ms, so about 530ms.

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.

4 Comparison to other protocols

4.1 Simplex

As already mentioned above in Section 3.4.2, the optimistic proposal-commit latency \((3\delta)\) and the optimistic consecutive proposal latency \((2\delta)\) 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 \( v \), 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.

4.2 HotStuff and HotStuff-2

We may also compare DispersedSimplex to HotStuff [YMR+18] and the recently proposed improvement HotStuff-2 [MN23].
4.2.1 Latency. HotStuff-2 has an optimistic proposal-commit latency of $5\delta$ while HotStuff has an optimistic proposal-commit latency of $7\delta$. Pipelined versions of these protocols can achieve an optimistic consecutive proposal latency $2\delta$. 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 $3\delta$ for DispersedSimplex).

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

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

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

Recall that $\beta$ bounds the block size, $\kappa$ the signature share/certificate size, and $\lambda$ 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 broadcast its proposed block to all parties — 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.

4.2.3 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 3.4.3. With the parameters we used there (1Gbps 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 $\beta$ 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 50ms 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 9.3MB for DispersedSimplex), the optimistic consecutive proposal latency is 1s (vs 430ms for DispersedSimplex), and (for HotStuff-2) the optimistic proposal-commit latency is that plus about 300ms, so about 1.3s (vs 530ms 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. Another factor to potentially consider is the fact that for a balanced protocol like DispersedSimplex, the rate at which each party is transmitting is fairly constant, while for protocols like HotStuff, it is very bursty.

4.2.4 A tension between time-outs. Another issue with HotStuff-2 is that in addition to a timeout analogous to the value \( \Delta \) used in DispersedSimplex, there is a waiting period \( \Delta' \) 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 time-out \( \Delta \)-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 \( \Delta \). However, to preserve liveness in HotStuff-2, the leader would have to adjust \( \Delta' \) as well as \( \Delta \). Unfortunately, if the leader’s value of \( \Delta' \) becomes too large relative the time-out \( \Delta \)-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 \( \Delta \)-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.

4.3 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 approve a proposal for slot \( v \) from 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.
5 Other topics

5.1 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 time-out pool contains time-out 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 Time-out and Commit Property, there can never be a time-out certificate for slot $v_{com}$. So the party can maintain two data structures.

- A doubly linked list of those slots in the range $\{v_{com}, \ldots, v - 1\}$ for which it does not have a time-out certificate, in order from lowest to highest.
- A lookup table from $\{v_{com}, \ldots, v - 1\}$ 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 time-out certificate appears for a slot in the range $\{v_{com}, \ldots, v - 1\}$, 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 time-out certificates for slots $v^* + 1, \ldots, v - 1$.

5.2 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 time-out pool contains time-out certificates for each slot $v' + 1, \ldots, v - 1$.

- **Moving on from bad blocks.** In the protocol, in managing the approved block pool, when a party reconstructs the payload and finds that it is bad (either ⊥ 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 time-out pool contains time-out 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 time-out 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.

Acknowledgments

Thanks to Benjamin Chan and Rafael Pass for helpful discussions on the Simplex protocol.
References


