MiniCast: Minimizing the Communication Complexity of Reliable Broadcast

Thomas Locher[1] Victor Shoup[2]

[1] DFINITY
thomas.locher@dfinity.org

[2] Offchain Labs
victor@shoup.net

April 14, 2024

Abstract. We give a new protocol for reliable broadcast with improved communication complexity for long messages. Namely, to reliably broadcast a message a message m over an asynchronous network to a set of n parties, of which fewer than $n/3$ may be corrupt, our protocol achieves a communication complexity of $1.5|m|n+O(\kappa n^2 \log(n))$, where κ is the output length of a collision-resistant hash function. This result improves on the previously best known bound for long messages of $2|m|n + O(\kappa n^2 \log(n))$.

1 Introduction

Reliable broadcast is a fundamental problem in distributed computing that was first introduced by Bracha [Bra87]. In this problem, we have n parties $P_1, \ldots, P_n$, at most t of which are corrupt, where $t < n/3$, connected by authenticated point-to-point channels (the bound $t < n/3$ is known to be optimal). The communication network is completely asynchronous (i.e., there is no bound on the time it takes to deliver messages) and the scheduling of message delivery is completely under the control of an adversary.

A **reliable broadcast** protocol allows a sender S to broadcast a single message m to $P_1, \ldots, P_n$. Such a protocol should satisfy the following **correctness** property (informally stated):

> *All honest parties that output a message must output the same message. Moreover, if the sender S is honest, that message is the one input by S.*

It should also satisfy the following **completeness** property (informally stated):

> *If any honest party outputs a message or an honest sender S inputs a message, then eventually all honest parties output a message.*

Here, "eventually" means if and when all protocol messages sent from honest parties to honest parties have been delivered.

In evaluating the efficiency of a reliable broadcast protocol, an important metric is *communication complexity*, which is defined as the total number of bits transmitted by all honest parties over the network in the worst case. Bracha's protocol has a communication complexity of $O(|m|n^2)$, where $|m|$ is the length (in bits) of the message m to be broadcast—essentially, every party echoes m to every other party. For very long messages, this communication complexity can be impractical.

The first improvement to the communication complexity of reliable broadcast was made by Cachin and Tessaro [CT05], who gave a protocol with communication complexity $O(|m|n +$

$\kappa n^2 \log(n)$), where κ is the output length of a hash function. Note that the security of their protocol relies on the collision resistance of this hash function (unlike Bracha's protocol, which makes no other assumptions beyond authenticated channels).

Subsequent work has primarily focused on improving the term $O(\kappa n^2 \log(n))$ (for example, see [DXR21,DXR22]). However, for long messages (i.e., when $|m| \gg \kappa n \log(n)$) the term $O(|m|n)$, will dominate the communication complexity, and the more interesting problem in this case is to reduce the constant in this term. The approach of Cachin and Tessaro is based on an erasure code that encodes m as a vector of fragments, any $n - 2t$ of which may be used to reconstruct m. For $t \approx n/3$, each fragment will be of size $\approx 3|m|/n$, and every party will echo one such fragment to all other parties, leading to a communication complexity of (at least) $\approx (3|m|/n) \cdot n^2 = 3|m|n$ for any protocol based on this approach.

The first work to improve the communication complexity for long messages is due to Locher [Loc24]. Instead of using an erasure code with a recovery threshold of $n - 2t$, Locher shows how to use a recovery threshold of $n - t$. Using this higher threshold presents a number of problems, which Locher shows how to overcome, resulting in a protocol whose communication complexity (for long messages) is $\approx 2|m|n$. Locher proves a lower bound of $\approx 1.5|m|n$ on the communication complexity for a certain restricted class of protocols, but leaves as open questions:

(i) Can we prove a general lower bound of $\approx 1.5|m|n$?
(ii) Is there a protocol whose communication complexity is $\approx 1.5|m|n$?

In this paper, we answer the second question affirmatively: we give a new protocol, called *MiniCast*, whose worst-case communication complexity is

$$1.5|m|n + O(\kappa n^2 \log(n)).$$

Protocol MiniCast enjoys several other properties:

- its *round complexity* in the "good case" (when the sender is honest) is 4;
- its communication is *well balanced*, meaning that each party transmits roughly the same amount of data (except for the sender, who transmits about twice as much);
- its security only relies on a collision-resistant hash function.

The good-case round complexity of Protocol MiniCast, which is 4, is not as good as that of Bracha's protocol, which is 3. To address this issue, we also give an "optimistic" variant of our protocol, called OptMiniCast. This protocol has a good-case round complexity of 3, and has a worst-case communication complexity of $\approx 2|m|n$. However, unless the sender *detectably* misbehaves, the communication complexity is still $\approx 1.5|m|n$. We discuss various strategies to mitigate against such misbehavior, so that in many practical settings, we can assume it is rare. We note that like Protocol OptMiniCast, Protocol $\mathcal{A}_{bit}$ in [Loc24] also has a good-case round complexity of 3 and a worst-case communication complexity of $\approx 2|m|n$.

2 Preliminaries

We assume n parties $P_1, \ldots, P_n$, at most t of which are corrupt, where $t < n/3$. We assume parties are connected by authenticated point-to-point channels. The communication network is completely asynchronous, and we assume that message delivery is completely under the control of the adversary.

That is, although the adversary cannot modify the contents of messages sent between honest parties, he can decide if and when they are delivered.

As stated in Section 1, a reliable broadcast protocol allows a sender S to broadcast a single message m to $P_1, \ldots, P_n$, and should satisfy a *correctness* and *completeness* property. We define these properties more precisely here.

The **correctness** property states that:

(i) any two honest parties that output a message must output the same message;
(ii) at the time any honest party outputs a message, if S is honest, then S must have previously input the same message.

The **completeness** property states that:

at any point in time where all messages that have been sent by some honest party to another honest party have been delivered, we have the following: if either
(i) some honest party has output a message, or
(ii) if S is honest and has input a message,
then all honest parties have output a message.

Of course, this definition of *completeness* is only meaningful for protocols with bounded message complexity—which will be the case for our protocols here.

Incorporating cryptography. The correctness and completeness claims about our protocols will be based on cryptographic assumptions (namely, the collision resistance of a hash function). To incorporate this into our model, we have to modify the above definitions of correctness and completeness. Specifically, each property is recast as a game played between an adversary and a challenger. The adversary encompasses the scheduling of message delivery as well as the behavior of the corrupt parties. If the sender is honest, the adversary also supplies the input to the sender. The challenger encompasses the behavior of the honest parties. For each game, we say the adversary *wins* the game if he can make the protocol execution violate the corresponding defining property. We say correctness (resp., completeness) holds if for every polynomial-time adversary wins the correctness (resp., completeness) game with at most negligible probability. Here, *polynomial-time* and *negligible* are defined in terms of an implicit *security parameter*.

Static vs adaptive corruptions. The above definitions and all of our results are stated in the *static corruption model*, where the adversary chooses which parties are corrupt at the very beginning of the attack. We work in this model mainly for simplicity. In fact, all of our protocols remain secure in the *adaptive corruption model*, where the adversary may choose which parties to corrupt during the course of the protocol execution.

Other notation. For a nonnegative integer k, we denote by $[k]$ the set $\{1, \ldots, k\}$.

3 Data encoding logic

Before presenting the detailed logic of the message flows of our protocol, we first present the basic logic of how data is encoded and how it flows through the protocol.

3.1 Erasure codes

For integer parameters $k \geq d \geq 1$, an (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 ℓ 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 ℓ.

Using a Reed-Solomon code, which is based on polynomial interpolation, we can realize a (k, d)-erasure code so that if $|m| = \ell$, then each fragment has size $\approx \ell/d$. 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 $\max(\lceil \ell/d \rceil, \lceil \log_2(k) \rceil)$—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

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

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).

Our reliable broadcast protocol uses two erasure codes.

- *An $(n, n-t)$-erasure code.*
  This is used to encode the message m to be broadcast by our protocol as a vector of n fragments, $f_1, \ldots, f_n$, any $n - t$ of which may be used to efficiently reconstruct the message m. If $|m| = \ell$, then the size of each fragment f_i, which is a function of ℓ, n, and t, is at most

$$\frac{\ell}{n-t} + O(\log(n)) \leq \frac{3}{2n}\ell + O(\log(n)).$$

- *An $(n, n-2t)$-erasure code.*
  This is used to encode a fragment f_i as above as a vector of n "mini-fragments" $\phi_{i1}, \ldots, \phi_{in}$, any $n - 2t$ of which may be used to reconstruct the fragment f_i. If $|m| = \ell$, then the size of each mini-fragment ϕ_{ij}, which is a function of ℓ, n, and t, is at most

$$\frac{\ell}{(n-t)(n-2t)} + O(\log(n)) \leq \frac{9}{2n^2}\ell + O(\log(n)).$$

3.2 Merkle trees

Our reliable broadcast protocol will also make use of 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 for v_i under r 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.

We shall actually make use of a two-level Merkle tree structure. For a given message m, the top level Merkle tree will be rooted at r and have leaves $r_1, \ldots, r_n$, where each r_i is itself the root of a Merkle tree whose leaves are the hashes of mini-fragments $\phi_{i1}, \ldots, \phi_{in}$ of the fragment f_i of m. We will therefore have validation paths π_i for r_i under r at position i, as well as validation paths π_{ij} for ϕ_{ij} under r_i at position j.

Each validation path π_i and π_{ij} will have size $O(\kappa \log(n))$, where κ is the output length of the hash function.

3.3 Encoding and decoding functions

Tags. For a given message length ℓ and Merkle tree root r, we define the **tag** $\tau := (\ell, r)$.

Certified fragments. For a tag $\tau = (\ell, r)$, 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 ℓ, and
- when we compute the mini-fragments $\phi_{i1}, \ldots, \phi_{in}$ for f_i and build the Merkle tree rooted at r_i whose leaves are the hashes of $\phi_{i1}, \ldots, \phi_{in}$, we find that π_i is a correct validation path for r_i under r at position i.

Certified mini-fragments. For a tag $\tau = (\ell, r)$, we shall call $(\phi_{ij}, \pi_{ij}, \pi_i)$ a **certified mini-fragment for τ at position** (i, j) if

- ϕ_{ij} has the correct length of a mini-fragment for a message of length ℓ,
- π_{ij} is a validation path for ϕ_{ij} under a root r_i, and
- π_i is a correct validation path for r_i under r.

We define several functions that will be used in our protocol. All of these functions take n and t as implicit parameters.

The Encode function. The first function, *Encode*, takes as as input a message m. It builds a two-level Merkle for m as above with root r (encoding m as a vector of fragments, encoding each fragment as a vector of mini-fragments, and then building a two-level Merkle tree whose leaves are the hashes of all of these mini-fragments). It returns

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

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

The Decode function. The second function, *Decode*, takes as input

$$\big(\ i,\ \tau,\ \{\{(f_j, \pi_j)\}_{j \in \mathcal{J}}\ \big),$$

where $i \in [n]$, $\tau = (\ell, r)$ is a tag, $\mathcal{J}$ is a subset of $[n]$ of size $n - t$, and each (f_j, π_j) is a certified fragment for τ at position j. It first reconstructs a message m' from the fragments $\{f_j\}_{j \in \mathcal{J}}$, using the size parameter ℓ. If $m' = \perp$, it returns $\perp$. Otherwise, it encodes m' as a vector of fragments,

encodes each f'_i as a vector of mini-fragments, and builds a two-level Merkle tree with root r' from the mini-fragments. If $r' \neq r$, it returns $\perp$. Otherwise, it returns

$$\big(\ m',\ \{(\phi'_{ji}, \pi'_{ji}, \pi'_j)\}_{j\in[n]}\ \big),$$

where each $(\phi'_{ji}, \pi'_{ji}, \pi'_j)$ is a certified mini-fragment for τ at position (j, i).

The Recover function. The third function, *Recover*, takes as input

$$\big(\ i,\ \tau,\ \pi_i,\ \{(\phi_{ij}, \pi_{ij})\}_{j\in\mathcal{J}}\ \big),$$

where $i \in [n]$, $\tau = (\ell, r)$ is a tag, $\mathcal{J}$ is a subset of $[n]$ of size $n - 2t$, and each $(\phi_{ij}, \pi_{ij}, \pi_i)$ is a certified mini-fragment for τ at position (i, j). It reconstructs and outputs the fragment f_i from the mini-fragments $\{\phi_{ij}\}_{j\in\mathcal{J}}$ using size parameter ℓ. (Note that the decoding procedure could output $\perp$, but in our application, this will not happen, except with negligible probability.)

Thomas: It may also happen if the assumption that $t < n/3$ is violated. Should we state that here as well?

3.4 The basic data flow

In our protocol, the sender will take a message m and compute

$$\big(\ \tau,\ \{(f_i, \pi_i)\}_{i\in[n]}\ \big) \leftarrow \mathit{Encode}(m)$$

and send to each party P_i the tag τ along with the certified fragment (f_i, π_i), and each such P_i will echo this tag and certified fragment to all parties. This will contribute at most

$$\tfrac{3}{2}\ell(n+1) + O(\kappa n^2 \log(n)) \tag{1}$$

to the communication complexity.

By obtaining a collection $\{(f_j, \pi_j)\}_{j\in\mathcal{J}}$ of $n - t$ certified fragments for the tag τ, party P_i can then compute

$$\mathit{Decode}\big(\ i,\ \tau,\ \{(f_j, \pi_j)\}_{j\in\mathcal{J}}\ \big).$$

The logic in *Decode* of re-encoding the reconstructed output message m' and comparing the new Merkle root r' to the Merkle root r in the tag τ ensures correctness in case of a corrupt sender. Also, the logic in *Decode* will generate mini-fragments so that P_i can send one certified mini-fragment to each other party that needs one. This will ensure that in the case of a corrupt sender, any party that has not obtained its own fragment from the sender will eventually be able to reconstruct that fragment from these mini-fragments using the function *Recover*. More precisely, after P_i has reconstructed an output message from $\{(f_j, \pi_j)\}_{j\in\mathcal{J}}$, it will send to each other party P_j a certified mini-fragment for τ at position (j, i). This will contribute at most

$$\tfrac{9}{2}\ell + O(\kappa n^2 \log(n)) \tag{2}$$

to the communication complexity. If P_j is able to collect $n - 2t$ such certified mini-fragments from $n - 2t$ distinct parties, then P_j can reconstruct its fragment f_j.

Combining (1) and (2), the total communication complexity is at most

$$\tfrac{3}{2}\ell(n+4) + O(\kappa n^2 \log(n)).$$

4 The main protocol

Our new reliable broadcast protocol, which we call Π_{MiniCast}, is presented in Fig. 1. We express the logic for the sender as a separate process, even though the sending party is also one of the receiving parties. For each receiving party, the main logic is presented as a loop, and in each loop iteration the party waits for one of several conditions to hold, triggering a reaction. These conditions are based on local state plus an implicit pool of received messages.

The details for data encoding are discussed in Section 3. At a high level, the protocol proceeds in rounds:

1. In the *disperse* round, the sender disseminates certified fragments with corresponding tags for its message.
2. In the *echo* round, each party echoes the tag it received from the sender in the *disperse* round to all other parties.
3. In the *vote* round, each party waits for $n-t$ messages from the *echo* round with tags that match the tag it received in the first round; when this happens, each party sends a *vote* message that contains its certified fragment and corresponding tag to all other parties.
4. In the *confirm* round, each party waits for $n-t$ messages from the *vote* round with matching tags; when this happens, each party P_i decodes to obtain an output message and sends a *confirm* message to each party P_j. This message contains the certified mini-fragment that P_j may need to construct its own fragment.
5. Upon collecting $n-t$ messages from the *confirm* round, each party delivers the output message decoded in the *confirm* round.

In addition, there is a "feedback loop", whereby a party P_i that did not receive an appropriate certified fragment from the sender in the *disperse* round can still generate a message in the *vote* round: if P_i receives $n-2t$ appropriate messages from the *confirm* round, these messages contain sufficiently many mini-fragments so as to allow P_i to recover its own fragment and send it to all other parties in the *vote* round.

The rough, intuitive argument for completeness is this. Suppose some honest party delivers an output message.

- This party must have received $n-t$ messages in the *confirm* round.
- Therefore, at least $n-2t$ will send out mini-fragments to any party that may need them.
- Therefore, every honest party will eventually send out a message in the *vote* round.
- Therefore, every honest party will eventually decode an output message and send out a message in the *confirm* round.
- Therefore, every honest party will eventually deliver an output message.

In addition to allowing each party to recover its "missing fragment", this "feedback loop" also plays a role similar to the "feedback loop" in Bracha's original protocol (whereby a "ready" message is generated upon receiving either $n-t$ "echo" messages or $t+1$ "ready" messages).

We also note that by having each party wait to receive sufficiently many *echo* messages with the same tag before sending out a *vote* message, we ensure each party will only send a *vote* message that contains a certified fragment for the "right" tag. This is only needed when dealing with a corrupt sender who may send different tags to different parties in the *disperse* round.

Π_{MiniCast}

// ***Sender*** S ***with input*** m
$(\tau, \{(\pi_i, f_i)\}_{i\in[n]}) \leftarrow \mathit{Encode}(m)$
for $i \in [n]$: send $(\mathsf{disperse}, \tau, \pi_i, f_i)$ to P_i

// ***Receiving party*** P_i
$\mathit{echoed} \leftarrow \mathit{voted} \leftarrow \mathit{confirmed} \leftarrow \mathit{delivered} \leftarrow \mathsf{false}, \quad \mathit{acquired} \leftarrow m^* \leftarrow \perp$

repeat forever:
  wait until either:
    not *echoed* and for some τ, f_i, π_i: received $(\mathsf{disperse}, \tau, f_i, \pi_i)$ from $S \Rightarrow$
      $\mathit{echoed} \leftarrow \mathsf{true}$
      if (f_i, π_i) is a certified fragment for τ at position i then
        $\mathit{acquired} \leftarrow (\tau, f_i, \pi_i)$
        send (echo, τ) to $P_1, \ldots, P_n$

    not *voted* and for some τ, f_i, π_i:
      $\mathit{acquired} = (\tau, f_i, \pi_i)$ and received (echo, τ) from $n - t$ distinct parties $\Rightarrow$
      $\mathit{voted} \leftarrow \mathsf{true}$
      send $(\mathsf{vote}, \tau, f_i, \pi_i)$ to $P_1, \ldots, P_n$

    not *voted* and for some τ, π_i, $\mathcal{J} \subseteq [n]$ of size $n - 2t$, and $\{(\phi_{ij}, \pi_{ij})\}_{j\in\mathcal{J}}$:
      for all $j \in \mathcal{J}$: received $(\mathsf{confirm}, \tau, \phi_{ij}, \pi_{ij}, \pi_i)$ from P_j
        and $(\phi_{ij}, \pi_{ij}, \pi_i)$ is a certified mini-fragment for τ at position $(i, j) \Rightarrow$
      $\mathit{voted} \leftarrow \mathsf{true}$
      $f_i \leftarrow \mathit{Recover}(i, \tau, \pi_i, \{(\phi_{ij}, \pi_{ij})\}_{j\in\mathcal{J}})$
      send $(\mathsf{vote}, \tau, f_i, \pi_i)$ to $P_1, \ldots, P_n$

    not *confirmed* and for some τ, $\mathcal{J} \subseteq [n]$ of size $n - t$, and $\{(f_j, \pi_j)\}_{j\in\mathcal{J}}$:
      for all $j \in \mathcal{J}$: received $(\mathsf{vote}, \tau, f_j, \pi_j)$ from P_j
        and (f_j, π_j) is a certified fragment for τ at position $j \Rightarrow$
      $\mathit{confirmed} \leftarrow \mathsf{true}$
      if $\mathit{Decode}(i, \tau, \{\{(f_j, \pi_j)\}_{j\in\mathcal{J}})$ returns $(m', \{(\phi'_{ji}, \pi'_{ji}, \pi'_j)\}_{j\in[n]})$ then
        $m^* \leftarrow m'$
        for $j \in [n]$: send $(\mathsf{confirm}, \tau, \phi'_{ji}, \pi'_{ji}, \pi'_j)$ to P_j

    not *delivered* and $m^* \neq \perp$ and received $(\mathsf{confirm}, \ldots)$ from $n - t$ distinct parties $\Rightarrow$
      $\mathit{delivered} \leftarrow \mathsf{true}$
      output m^*

Fig. 1. The MiniCast reliable broadcast protocol

Theorem 4.1. *Assuming collision resistance of the underlying hash functions, Protocol Π_{MiniCast} satisfies the* correctness *and* completeness *property (with overwhelming probability, for any polynomial-time adversary).*

Proof. We start with some terminology. We say:

- an honest party *echoes* τ if it sends a message (echo, τ);
- an honest party *votes for* τ if it sends a message $(\mathsf{vote}, \tau, \ldots)$;
  - we say it *votes by echo* if it does so because it received $n - t$ echo messages;
  - we say it *votes by confirmation* if it does so because it received $n - 2t$ $\mathsf{confirm}$ messages;
- an honest party *confirms* τ if it sends a message $(\mathsf{confirm}, \tau, \ldots)$.

We can also extend these definitions to corrupt parties, saying that a corrupt party *echoes*, *votes for*, or *confirms* τ if any honest party receives a corresponding message from that corrupt party.

The following properties prove the theorem.

P1: If an honest party confirms τ, then at least $n - t$ parties previously echoed τ.

Proof. Consider the point in time where the first honest party confirms τ. Some honest party must have voted for τ, and this must be a vote by echo (since no honest party previously confirmed τ and $n - 2t > t$).

P2: If an honest party votes for τ, then at least $n - t$ parties previously echoed τ.

Proof. If this is a vote by echo, then the claim holds by definition. If this is a vote by confirmation, then at least one of these confirmations must have come from an honest party (since $n - 2t > t$), and the property follows from P1.

P3: Any honest parties that vote or confirm a tag, must vote or confirm the same tag.

Proof. The property follows from P1 and P2 and the fact that each honest party echoes at most one tag (quorum intersection argument).

P4: If any honest party delivers an output message, then all honest parties eventually deliver the same output message (with overwhelming probability, assuming collision resistance).

Proof. Consider the point in time where the first honest party delivers a message. It does so because it received $n - t$ $\mathsf{confirm}$ messages, of which $n - 2t$ must be from honest parties. This party must have also confirmed some tag τ, and by P3, all honest parties that confirm or vote must confirm or vote for the same tag τ. Since all honest parties eventually receive at least $n - 2t$ $\mathsf{confirm}$ messages, all honest parties will eventually either vote for τ by echo or vote for τ by confirmation.

By collision resistance, all honest parties that generate or receive a certified fragment or mini-fragment for τ at the same location will generate or receive the same fragment or mini-fragment. In particular, if one honest party collects $n - t$ certified fragments for τ and then proceeds to set m^* to m' and confirm τ, then any other honest party that collects $n - t$ certified fragments for τ will also confirm τ and set m^* to m'. In addition, if one honest party P_i collects $n - 2t$ certified mini-fragments for τ at positions $(i, \cdot)$, then one of these mini-fragments must be from an honest party P_j, and the mini-fragments received by P_i must decode to the same fragment f_i that P_j generated.

Therefore, all honest parties will eventually obtain the $n - t$ vote messages they need to confirm τ, and so will also eventually send and receive $n - t$ $\mathsf{confirm}$ messages.

P5: If the sender is honest and initiates the protocol, then (i) all honest parties will eventually output a message, (ii) honest parties output a message only after the sender has initiated the protocol and that message is the sender's input message m (with overwhelming probability, assuming collision resistance).

Proof. Suppose the sender initiates the protocol and let τ be the tag generated by the sender. By P1, any honest party that confirms must confirm τ and by P2, any honest that votes must vote for τ. Since every honest party will eventually echo τ, every honest party will eventually vote for τ. By collision resistance, all honest parties that generate or receive a certified fragment or mini-fragment for τ at the same location will generate or receive the same fragment or mini-fragment. When any honest party collects $n-t$ certified fragments for τ, again by P1, the sender must have already initiated the protocol, and these fragments must match those generated by the sender. Therefore, this honest party will set m^* to m and confirm τ. So eventually, all honest parties will confirm τ and deliver the output message m.

That completes the proof. □

4.1 Message and communication complexity

The message complexity of Π_{MiniCast} is obviously $O(n^2)$.

As sketched in Section 3, the communication complexity of Π_{MiniCast} is at most

$$\tfrac{3}{2}\ell(n+4) + O(\kappa n^2 \log(n)), \tag{3}$$

where κ is the output length of the hash function. Here, ℓ represents a bound on the message length. When the sender S is honest, we can take ℓ to be the length of the message m that S is sending. When S is corrupt, we can take ℓ to be the value that is in the dispersed tag τ—honest parties will not transmit any data until the tag is agreed upon, and so this value is well defined, and the size of the fragments and mini-fragments transmitted by the honest parties will be consistent with a message of length ℓ (and if any honest party ultimately delivers a message, it will be of length ℓ). Any particular application may impose a bound on the message length and the honest parties can enforce this bound when they execute the protocol.

We also note that the communication is well balanced among all parties—each party transmits roughly the same amount of data, except for the sender, which transmits about twice as much.

4.1.1 Minor improvements to the communication complexity.

The reader will note that the above stated communication complexity (3) does not quite match the bound reported in Section 1, namely

$$\tfrac{3}{2}\ell n + O(\kappa n^2 \log(n)) \tag{4}$$

This better bound can be easily achieved through the following simple optimizations:

(i) A message sent from a party to itself need not be transmitted over the network.
(ii) In the *vote* round, each party may omit the certified fragment from *vote* messages sent to the sender.
(iii) In the *confirm* round, each party may omit the certified mini-fragment from *confirm* messages sent to the $n-t$ parties from which a *vote* message was received.

With these optimizations:

- In the *disperse round*, the sender transmits $n-1$ certified fragments.
- In the *vote* round, the sender transmits $n-1$ certified fragments, while the $n-1$ other parties each transmit $n-2$ certified fragments.
- In the *confirm* round, each party transmits at most $t < \frac{1}{3}n$ mini-fragments.

When applying these optimizations, $n^2 - n$ certified fragments and $\frac{1}{3}n^2$ certified mini-fragments are transmitted altogether, from which the bound (4) follows.

4.2 Computational cost

The computational cost is dominated by the following operations, which are performed at most once per party:

- encode the message into n fragments;
- encode all n fragments into n^2 mini-fragments;
- decode $n-t$ fragments to recover message;
- decode $n-2t$ mini-fragments to recover a single fragment;
- hash n^2 mini-fragments.

4.3 Storage costs

As we have described the protocol, each party stores all protocol messages sent to it by any party. However, an obvious optimization that one would apply in any practical implementation is to notice that any honest party transmits at most one message of any given "type" (disperse, echo, vote, confirm). Therefore, each party need only store one message of each type that it receives from any one party. Thus, the number of messages that a party stores is $O(n)$.

The total space occupied by these messages is at most $O(\ell n + \kappa n^2 \log(n))$. Here, ℓ represents a bound on the message length. However, unlike in Section 4.1, we will simply have to take ℓ to be some application-dependent bound on message length, which may be enforced by honest parties.

4.4 A concrete example

Suppose messages are of size 4MB, $n = 100$, and $t = 33$. Each fragment is about 60KB and each mini-fragment is about 1.8KB. Assuming 32-byte hash outputs, the size of the Merkle validation paths is about 0.25KB. In this case, the communication complexity increases by 0.25KB per certified fragment and 0.5KB per certified mini-fragment because the latter requires two validations paths.

Using the optimizations discussed in Section 4.1.1, the total communication complexity is roughly

$$(100^2 - 100) \cdot (60\text{KB} + 0.25\text{KB}) + 100 \cdot 33 \cdot (1.8\text{KB} + 0.5\text{KB}) \approx 597\text{MB},$$

which is in fact a bit less than the asymptotic estimate of 600MB corresponding to $\frac{3}{2}\ell n$. This estimate only counts the contribution to the communication complexity made by the certified fragments and mini-fragments. The communication complexity contributed by all other data may increase this bound by a few MB.

Thomas: How do you get the number 597 MB? I get 604065 KB (KiB?). If it's KB, it would be 604 MB. If it's KiB, it would be roughly 590 MiB.

We also estimate the running time for various computations.

For the erasure coding, we used the `reed-solomon-simd` library[3], which is based on [LC12,LAHC16]. We benchmarked this implementation on a Macbook Pro with an Apple M1 Max CPU. We found the following:

- time to encode message into 100 fragments: 1.4ms;
- time to encode all 100 fragments into 100^2 mini-fragments: 3.9ms;
- time to decode 67 fragments to recover message: 6.5ms;
- time to decode 34 mini-fragments to recover a single fragment: 0.6ms.

Based on these numbers, all encoding and decoding operations will take at most about 13ms.

As for hashing, using `openssl speed sha256`, we estimate the time to hash 100^2 mini-fragments on the same Macbook Pro to be about 12ms.

So the total computation time is approximately 25ms. Note that this time could conceivably be reduced somewhat using multiple cores. Let us compare this computation time to time spent in communications. Each party will send about 6MB of data (but the sender will send twice this amount, but at different times in the protocol). Assuming a 1Gbps network speed, transmitting 6MB of data will take about 48ms. In the "good case", there will be 4 rounds of communication. The network latency per round obviously depends on geography and many other factors, but we can estimate a range of 10-100ms per round in a typical wide-area network configuration.

Let us compare these estimates to the time it takes for the sender to naively send a 4MB message to all 100 parties separately. Again using the 1Gbps network speed estimate, it will take the sender 3.2 *seconds* just to transmit all of this data. In contrast, even using the upper range of 100ms communication latency per round, our protocol should take a total of less than 0.5 seconds. Of course, as an alternative, one may also consider using some sort of "gossip protocol" to disseminate the message more efficiently—but that will likely cost just as much or more in network latency as our protocol.

These estimates should at least convince the reader that erasure-code-based broadcast protocols in general, and our protocol in particular, are of more than just theoretical interest. From a practical perspective, perhaps the most important feature of erasure-code-based broadcast protocols is the fact that communication is typically well balanced among all parties.

5 An optimistic 3-round variant

We now present an "optimistic" version of our protocol Π_{MiniCast} that has better round complexity (just 3 rounds in the "good case" where the sender is honest, instead of 4). The modification is simple: instead of broadcasting an *echo* message (containing just a tag) when receiving certified fragment from the sender, a *vote* message (containing the certified fragment and tag) is broadcast right away. This variation, which we call $\Pi_{\text{OptMiniCast}}$, is presented in Fig. 2. Even though much of the protocol is identical to the original, we present the protocol in its entirety, using color to highlight what is old and what is new.

Note that a receiving party may vote twice if the sender is corrupt. Because of this, the worst-case communication complexity of $\Pi_{\text{OptMiniCast}}$ is worse than that of Π_{MiniCast}. We shall discuss this issue below in Section 5, including mitigation strategies that in many settings will ensure that this "double voting" rarely, if ever, occurs.

[3] See `https://github.com/AndersTrier/reed-solomon-simd`.

$\Pi_{\text{OptMiniCast}}$

// ***Sender*** S ***with input*** m
$(\tau, \{(\pi_i, f_i)\}_{i\in[n]}) \leftarrow \mathit{Encode}(m)$
for $i \in [n]$: send $(\mathsf{disperse}, \tau, \pi_i, f_i)$ to P_i

// ***Receiving party*** P_i
d-voted $\leftarrow$ *c-voted* $\leftarrow$ *confirmed* $\leftarrow$ *delivered* $\leftarrow$ false, $\tau^* \leftarrow m^* \leftarrow \perp$

repeat forever:
  wait until either:
    not (*d-voted* or *c-voted*) and for some τ, f_i, π_i: received $(\mathsf{disperse}, \tau, f_i, \pi_i)$ from $S \Rightarrow$
      d-voted $\leftarrow$ true
      if (f_i, π_i) is a certified fragment for τ at position i then
        $\tau^* \leftarrow \tau$
        send $(\mathsf{vote}, \tau, f_i, \pi_i)$ to $P_1, \ldots, P_n$

    not *c-voted* and for some $\tau \neq \tau^*$, π_i, $\mathcal{J} \subseteq [n]$ of size $n - 2t$, and $\{(\phi_{ij}, \pi_{ij})\}_{j\in\mathcal{J}}$:
      for all $j \in \mathcal{J}$: received $(\mathsf{confirm}, \tau, \phi_{ij}, \pi_{ij}, \pi_i)$ from P_j
        and $(\phi_{ij}, \pi_{ij}, \pi_i)$ is a certified mini-fragment for τ at position $(i, j) \Rightarrow$
      c-voted $\leftarrow$ true
      $f_i \leftarrow \mathit{Recover}(i, \tau, \pi_i, \{(\phi_{ij}, \pi_{ij}))\}_{j\in\mathcal{J}})$
      send $(\mathsf{vote}, \tau, f_i, \pi_i)$ to $P_1, \ldots, P_n$

    not *confirmed* and for some τ, $\mathcal{J} \subseteq [n]$ of size $n - t$, and $\{(f_j, \pi_j)\}_{j\in\mathcal{J}}$:
      for all $j \in \mathcal{J}$: received $(\mathsf{vote}, \tau, f_j, \pi_j)$ from P_j
        and (f_j, π_j) is a certified fragment for τ at position $j \Rightarrow$
      confirmed $\leftarrow$ true
      if $\mathit{Decode}(i, \tau, \{\{(f_j, \pi_j)\}_{j\in\mathcal{J}})$ returns $(m', \{(\phi'_{ji}, \pi'_{ji}, \pi'_j)\}_{j\in[n]})$ then
        $m^* \leftarrow m'$
        for $j \in [n]$: send $(\mathsf{confirm}, \tau, \phi'_{ji}, \pi'_{ji}, \pi'_j)$ to P_j

    not *delivered* and $m^* \neq \perp$ and received $(\mathsf{confirm}, \ldots)$ from $n - t$ distinct parties $\Rightarrow$
      delivered $\leftarrow$ true
      output m^*

Fig. 2. An optimistic variant of the MiniCast reliable broadcast protocol

Theorem 5.1. *Assuming collision resistance of the underlying hash functions, Protocol $\Pi_{\text{OptMiniCast}}$ satisfies the* correctness *and* completeness *property (with overwhelming probability, for any polynomial-time adversary).*

In addition, if the sender is honest, then no honest party votes more than once; otherwise, the number of honest parties that vote twice is at most t.

Proof. We use the same terminology introduced in the proof of Theorem 4.1 to say that a party *votes for* or *confirms* a tag τ. However, we now distinguish between a *d-vote*, where an honest party votes in response to a disperse message from the sender, an a *c-vote*, where it votes in response to $n - 2t$ confirm messages. Note that an honest parties may vote twice, but it d-votes at most once and it c-votes at most once.

The following properties prove the theorem.

P1: If an honest party confirms τ, then there is a set of at least $n - t$ parties that previously voted for τ such all of the honest parties in this set sent a d-vote for τ.

Proof. Consider the point in time where the first honest party confirms τ. That party must have received $n - t$ votes for τ. Moreover, among these votes, those coming from honest parties must be d-votes (since no honest party previously confirmed τ and $n - 2t > t$).

P2: Any honest parties that confirm a tag, must confirm the same tag.

Proof. The property follows from P1 and the fact that each honest party d-votes at most once (quorum intersection argument).

P3: If any honest party c-votes for a tag, some honest party must have confirmed that tag; in particular, any honest parties that c-vote or confirm a tag, must c-vote or confirm the same tag.

Proof. In any c-vote, at least one of the confirmations must have come from an honest party (since $n - 2t > t$). The rest follows from P2.

P4: If any honest party delivers an output message, then all honest parties eventually deliver the same output message (with overwhelming probability, assuming collision resistance).

Proof. Consider the point in time where the first honest party delivers a message. It does so because it received $n - t$ confirm messages, of which $n - 2t$ must be from honest parties. This party must have also confirmed some tag τ, and by P3, all honest parties that confirm or c-vote must confirm or c-vote for the same tag τ. Thus, all honest parties will eventually c-vote for τ unless they have already d-voted for τ. So in any case, all honest parties will eventually vote for τ.

The rest of the argument proceeds essentially as in the proof of P4 in Theorem 4.1. In particular, all honest parties will eventually obtain the $n - t$ vote messages they need to confirm τ; however, we must argue that no honest party did not already receive $n - t$ votes for a different tag. This follows from P6 below—if it could also garner $n - t$ votes for a different tag, at least $n - 2t > t$ of these votes must be from honest parties.

P5: If the sender is honest and initiates the protocol, then (i) all honest parties will eventually output a message, (ii) honest parties output a message only after the sender has initiated the protocol and that message is the sender's input message m (with overwhelming probability, assuming collision resistance).

Proof. Suppose the sender initiates the protocol and let τ be the tag generated by the sender. Certainly, any party that d-votes must d-vote for τ. By P1 and P3, any honest party that

confirms or c-votes must confirm or c-vote for τ. Every honest party will eventually d-vote or c-vote, and so will eventually d-vote or c-vote for τ.

The rest of the argument proceeds as in the proof of P5 in Theorem 4.1.

P6: If the sender is honest, then no honest party votes more than once; otherwise, the number of honest parties that vote twice is at most t.

Proof. Suppose the sender is honest and an honest party P_i votes twice, which means that it must first d-vote and then c-vote. When it c-votes for a tag, by P3, some honest party must have confirmed that tag, and so by P1, some honest parties must have d-voted for that tag. But because the sender is honest, that tag must be the same as the tag for which P_i already d-voted—and so P_i would not have c-voted, a contradiction.

Suppose the sender is corrupt and some set S of $t+1$ honest parties vote twice. By P3, all c-votes are for the same tag τ which must have been confirmed by some honest party. Each party in S must first d-vote for some tag different from τ and then c-vote for τ. By P1, there is a set T of $n-t$ parties that voted for τ, and moreover, all of the honest parties in T d-voted for τ. The sets S and T must overlap, and so one of the honest parties in S must have d-voted for τ, a contradiction.

That completes the proof. □

5.1 Message and communication complexity

The message complexity of $\Pi_{\text{OptMiniCast}}$ is still $O(n^2)$.

If the sender is honest, then by Theorem 5.1, no honest parties vote more than once, and it follows that the worst-case communication complexity of $\Pi_{\text{OptMiniCast}}$ in this case is at most

$$\tfrac{3}{2}\ell n + O(\kappa n^2 \log(n)),$$

where ℓ is the length of the message m that S is sending. Here, we are applying the same optimizations discussed in Section 4.1.1.

If the sender is corrupt, then by Theorem 5.1, at most $t < n/3$ honest parties vote more than once, and so the total number of vote messages transmitted by honest parties is at most $\frac{4}{3}n$. It follows that the worst-case communication complexity of $\Pi_{\text{OptMiniCast}}$ in this case is at most

$$2\ell n + O(\kappa n^2 \log(n)).$$

Here, ℓ represents a bound on the message length. Unlike in Section 4.1, honest parties may transmit fragments before a tag is agreed upon, and we will simply have to take ℓ to be some application-dependent bound on message length, which may be enforced by honest parties.

5.2 Computational cost

The computational cost of $\Pi_{\text{OptMiniCast}}$ is roughly the same as that of Π_{MiniCast}.

5.3 Storage costs

In Section 4.3, we noted that in Π_{MiniCast} an honest party only transmits at most one message of a given "type", and therefore, a party need only store one message of each type that it receives from

any one party. In $\Pi_{\text{OptMiniCast}}$, however, an honest party may send two vote messages. Therefore, each party must store up to two vote messages that it receives from any one party. Thus, the number of messages that a party stores is still $O(n)$.

The total space occupied by these messages is at most $O(\ell n + \kappa n^2 \log(n))$, where ℓ is some application-dependent bound on message length, which may be enforced by honest parties.

5.4 Dealing with corrupt senders

As we have seen, the worst-case communication complexity of $\Pi_{\text{OptMiniCast}}$ is at most

$$\tfrac{3}{2}\ell n + O(\kappa n^2 \log(n)),$$

unless the sender is corrupt and makes some honest party vote twice.

We suggest two simple ways of dealing with this.

5.4.1 Ignoring corrupt senders. In a long running system where many instances of the reliable broadcast protocol are executed over time, if a party P sends two vote messages in any one instance of the protocol, P can be sure that the sender S is corrupt. Therefore, P can simply ignore all messages received from S in any future executions of the protocol; in particular, P will never vote twice in any such future execution when S is the sender (while P may "c-vote" it will never "d-vote"). In many applications of reliable broadcast, although a party may be engaged in many reliable broadcast protocol instances simultaneously, it will be engaged in at most one instance with any given sender at any point in time. In such applications, over the lifetime of the system, there will be at most $O(n^2)$ "extra" votes in total that are ever sent by any honest parties (each honest party will send an "extra" vote at most once for each corrupt sender).

5.4.2 Implicating corrupt senders. Suppose we modify $\Pi_{\text{OptMiniCast}}$, replacing each tag $\tau = (\ell, r)$ throughout the protocol with $\tau = (\ell, r, \sigma)$, where σ is a signature under the public key of the sender S on a message that says "S is dispersing (ℓ, r) in protocol instance id". Here, id is an identifier that identifies the particular instance of the protocol. Each party needs to verify the signature when it processes a disperse message, but need not verify the signature in processing other protocol messages.

If a party does vote twice in a given protocol instance, it will be able to form an "equivocation proof", consisting of the two tags, and broadcast this equivocation proof to all parties (and even external parties). This equivocation proof provides irrefutable proof that S was corrupt. In response to this proof, various actions can be taken. At the very least, all parties can ignore all messages sent by S going forward. Additionally, S could be forcefully removed from the committee and/or punished in some way, financially or legally.

6 Conclusion

The proposed reliable broadcast protocols achieve a communication complexity of $\approx 1.5|m|n$ for long messages, resolving the open question whether a bound better than $\approx 2|m|n$ can be attained. Since each party must receive at least $|m|$ bits, it is evident that the communication complexity is close to optimal. A lower bound of $1.5|m|n$ has been shown for any 3-round protocol even if

communication occurs in synchronous rounds and at most $t < n/3$ parties exhibit crash failures (see [Loc24]). Protocol $\Pi_{\text{OptMiniCast}}$ achieves the same asymptotic bound (for sufficiently long messages) in 3 rounds even if communication is asynchronous and t parties can exhibit arbitrary corrupt behavior aside from equivocation. While tight bounds have not been shown, we conjecture that no better bound than $1.5|m|n$ can be achieved quite generally, i.e., protocol Π_{MiniCast} minimizes the communication complexity of reliable broadcast.

References

Bra87. G. Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.*, 75(2):130–143, 1987.

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.

DXR21. S. Das, Z. Xiang, and L. Ren. Asynchronous Data Dissemination and its Applications. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2705–2721. ACM, 2021. Also available at https://eprint.iacr.org/2021/777.

DXR22. S. Das, Z. Xiang, and L. Ren. Near-optimal Balanced Reliable Broadcast and Asynchronous Verifiable Information Dispersal. Cryptology ePrint Archive, Report 2022/052, 2022. https://ia.cr/2022/052.

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.

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