OPTIMUMP2P: Fast and Reliable Gossiping in P2P Networks

Nicolas Nicolaou[1], Onyeka Obi[1], Aayush Rajasekaran[1], Alejandro Bergasov[1], Aleksandr Bezobchuk[1], Kishori M. Konwar[1], Michael Meier[1], Santiago Paiva[1], Har Preet Singh[1], Swarnabha Sinha[1], Sriram Vishwanath[2], and Muriel Médard[1]
[1]{nicolas,oni,aayush,alejandro,bez,kkonwar,mike, santiago,harpreet,swarna,mmedard}@getoptimum.xyz
[1]Optimum, Cambridge, MA, USA
[2]sriram@utexas.edu
[2]University of Texas at Austin, TX, USA

***Abstract*—Gossip algorithms are pivotal in the dissemination of information within decentralized systems. Consequently, numerous gossip libraries have been developed and widely utilized especially in blockchain protocols for the propagation of blocks and transactions. A well-established library is *libp2p*, which provides two gossip algorithms: *floodsub* and *gossipsub*. These algorithms enable the delivery of published messages to a set of peers. In this work we aim to enhance the performance and reliability of *libp2p* by introducing OPTIMUMP2P, a novel gossip algorithm that leverages the capabilities of Random Linear Network Coding (RLNC) to expedite the dissemination of information in a peer-to-peer (P2P) network. Preliminary research from the Ethereum Foundation has demonstrated the use of RLNC in the significant improvement in the block propagation time [15]. Here we present extensive evaluation results both in simulation and real-world environments that demonstrate the performance gains of OPTIMUMP2P over the Gossipsub protocol.**

I. INTRODUCTION

Gossip algorithms, also known as epidemic protocols [5], [12], [24], are a class of decentralized communication strategies used in distributed systems to disseminate information efficiently and robustly across a network. In these algorithms, nodes periodically exchange information with a randomly selected subset of neighboring nodes, mimicking the spread of gossip in social networks. In a sense, this is also strongly related to epidemics, by which a disease is spread by infecting members of a group, which in turn can infect others. This probabilistic approach ensures that information propagates rapidly and reliably, even in large-scale or dynamically changing networks, without requiring centralized coordination or global knowledge of the system topology. These characteristics make gossip algorithms inherently fault-tolerant, scalable, and adaptable, making them ideal for applications such as distributed databases, consensus protocols, and peer-to-peer networks.

Consequently, gossip protocols attracted the attention and were widely adopted in implementations of blockchain systems. They serve an efficient and reliable solution for various tasks, including transaction and block propagation as seen in Bitcoin [34] and Ethereum [8], peer discovery (e.g.,Ethereum [8]), reaching consensus (e.g., Tendermint [7]), and state synchronization (e.g., Hyperledger Fabric [4]) among others.

However, blockchain implementations often operate on top of a permissionless, asynchronous, message-passing network, susceptible to unpredictable delays and node failures. Therefore, improper use of gossiping approaches may lead to high network overhead and congestion, high propagation latencies, and erroneous information propagation due to message alteration by malicious actors. *libp2p* [27], is one of the latest network communication frameworks and gossip algorithms used in modern blockchain solutions like Ethereum 2.0 [14]. *libp2p* adopts two different push gossiping algorithms: ***Floodsub*** and ***Gossipsub***.

Floodsub, uses a flooding strategy where every node forwards messages to all of its neighbors. Although very efficient in discovering the shortest path and very robust in delivering a message to all the peers in the network, Floodsub suffered from bandwidth saturation and unbounded degree flooding.

Gossipsub is the successor of Floodsub, which addressed the shortcomings of the initial algorithm by organizing peers into topic-based mesh, network overlay, with a target mesh degree D and utilizing control messages for reducing message duplication. Briefly, the Gossipsub protocol works as follows. A publisher selects D peers among its peers and broadcasts its message to them. Each peer receiving a message performs preliminary validation and rebroadcasts the message to another D peers. Peers exchange control messages such as IWANT, IHAVE or IDONTWANT to inform their peers about their status regarding the propagation of a particular message. These enhancements enabled Gossipsub to reduce bandwidth usage, but the introduction of the bounded degree D increased the number of hops a message required to reach distant peers, resulting in higher delivery latencies. Furthermore, similar to the Floodsub protocol, each peer forwards the full message to its peers even when other peers may already have received the full message, also suffering from (reduced compared to Floodsub) message duplication.

So can we introduce a gossip protocol that is light in network usage and yet fast in information diffusion?

An idea to leverage coding in network gossip was proposed

978-3-903176-75-1 ©2025 IFIP

in [38]. Luby Transform (LT) [31] codes were used that stream algebraically coded elements from a single source to multiple destinations. Although efficient in direct multicast and shallow network topologies, the performance of LT degrades when used in highly decentralized and multi-hop topologies where the central code generator may need to retransmit critical coded elements that may be lost during multi-hop relays.

In this work we propose the use of **Random Linear Network Coding (RLNC)** [21] for message broadcast. RLNC is a technique used in communication networks to enhance data transmission efficiency and robustness. In RLNC, data packets are encoded as random linear combinations of original packets over a finite field, typically $\mathbb{F}_{2^m}$. This approach allows intermediate nodes in the network to mix packets without decoding (aka *recode*), and the receiver can recover the original data by solving a system of linear equations once enough linearly independent combinations are received. RLNC leverages the algebraic properties of finite fields, that are deeply rooted in *Galois theory* [29], to ensure that the encoding and decoding processes are both efficient and probabilistically reliable. Using RLNC for gossip was introduced in [10], [11], which showed that optimum $O(n)$ dissemination of k messages is possible for both pull and push. The analysis was refined by [6], which considered pull, push and exchange, using Jackson networks with network coding [22] in a manner akin to [32]. Other settings, such as nodes with mobility [20], [36], [39], broadcast edges (equivalent to hyperedges) [13], [17] or correlated data [9], have been considered. Some initial results with large transfer of files were reported in [30]. Probably the most significant results are those that have shown, by using projection analysis [18] to consider the stopping time of gossip with RLNC, that, beyond order optimality in n, RLNC gossip achieves "perfect pipelining" [19]. The stopping time converges with high probability in optimal time, namely in time of $O(k+T)$, where k is the number of messages and T the dissemination time of a single message. Note that the general problem of network coding dissemination is hard to analyze when we do not use a large field size [16].

While the above results point to the potential benefit for using RLNC in gossip, in order for RLNC to be deployed in current decentralized systems, it requires the design of a full protocol. OPTIMUMP2P is a novel gossip mechanism based on RLNC, hence the term *Galois Gossip*, which significantly enhances the spread of information across the network. More precisely, as any network coding algorithm, RLNC allows the publisher to split a message into coded fragments (*shards*) and send a subset (or a linear combination) of shards – instead of the full message – to each of its peers. In turn, peers can forward linear combinations of shards they receive to their own peers. This approach has dual benefit:

1) **Faster Network Coverage:** it allows each peer to reach more peers for the same amount of data sent in full message counterparts (e.g., Gossipsub), and
2) **Message Duplication Reduction:** as peers receive different shards in parallel from different peers which combine to decode the original message.

Essentially, OPTIMUMP2P allows peers to spread information *piece by piece*, as oppose to traditional gossip approaches that broadcast full information between any pair of peers. Overall OPTIMUMP2P is a new protocol implemented within *libp2p* aiming to decrease latency, enhance fault tolerance, and optimize bandwidth usage. In the rest of the document we present the OPTIMUMP2P protocol and extensive experiments we conducted to compare the protocol's performance with Gossipsub, both in a simulation and real-world environments.

II. SYSTEM MODEL

OPTIMUMP2P aims to built a gossip service on top of a set of asynchronous, message-passing, network processes, we refer to as *peers*, a subset of which may fail arbitrarily. Each peer has a unique identifier from a set $\mathcal{I}$ and has access to a local clock which is not synchronized across peers.

Gossip Service: We assume a gossip service where peers may perform two primitive operations: (i) $\mathsf{publish}(m)_p$ operation where a peer $p \in \mathcal{I}$ requests the dissemination of a message m among the peers in $\mathcal{I}$, and (ii) a $\mathsf{deliver}(m)_p$ operation that delivers a message m to a peer $p \in \mathcal{I}$. From a user point of view a Gossip Service is defined by the following properties.

- *Validity*: if a peer publishes a message m, then m is eventually delivered at every correct peer.
- *Integrity*: a message m is delivered by a peer, if and only if m was previously published by some peer.

Communication Graph: Peers communicate through *asynchronous* channels. We assume two primary types of channels: reliable and unreliable. We represent our communication network by a directed graph $G = (V, E)$, where $V \subseteq \mathcal{I}$ is the set of vertices, representing the set of peers that participate in the service, and E the set of edges, or a set of links such that information can be reliably communicated from peer u to v for each $(u, v) \in E$. Each link e, $e \in E$ is associated with a non-negative number w_e representing the transmission capacity of the link in bit per unit time. The nodes u and v are referred to as *origin* and *destination*, respectively, of the link $(u, v) \in E$.

Messages: Each published message gets a unique identifier from a set $\mathcal{M}$, and contains a stream of bytes we refer to as the content of the message with a value $\mathbf{v} \in \mathcal{V}$.[1] We consider the use of *collision-resistant* cryptographic hash function which we denote by $\mathcal{H} : \{0,1\}^* \to \{0,1\}^n$ [23] for the generation of message identifiers in $\mathcal{M}$. A $\mathsf{publish}$ operation aims to propagate the contents of a message $m \in \mathcal{M}$, while a $\mathsf{deliver}$ operation aims to retrieve the contents of m and return them to the receiving peer.

Encoding/Decoding with RLNC: We use Random Linear Network Codes (RLNC) over a finite field $\mathbb{F}_{2^q}$, to encode the contents of a message $m \in \mathcal{M}$. In particular, for a given parameter k, a peer encodes the contents $\mathbf{v} \in \mathcal{V}$, using RLNC, to $k * p$ coded elements (or *shards*), for some $p \geq 1$.

[1]Note that the contents of a message can also be made unique by adding a random number from a large prime field, e.g., $\mathbb{F}_{2^q}$

Subsequently, the encoder or any other peer in the network may linearly combine any subset of shards to derive new linear combinations (i.e., new shards). Any k of the generated shards is sufficient to decode the value $\mathbf{v}$. We assume that k does not vary from message to message. For encoding, we do the following: (i) divide $\mathbf{v}$ into a vector of k elements $(v_1, v_2, \ldots, v_k)$; (ii) select a matrix $\mathbf{A}$ of coefficients at random from the finite field $\mathbb{F}_{2^q}$ such that $\mathbf{A}$ may be composed of $k * p$ rows and k columns; and (iii) multiply $\mathbf{A}$ with $(v_1, v_2, \ldots, v_k)$ to generate a vector of $\mathbf{c} = (c_1, c_2, \cdots, c_{k*p})$ of $k * p$ elements. The multiplication is formally illustrated below:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,k} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,k} \\ \vdots & \vdots & \ldots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{k*p,k} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{k*p} \end{pmatrix} \quad (1)$$

A node v communicates the coded shards $c_1, c_2, \cdots c_{k*p}$ to its neighbors. The matrix $\mathbf{A}$, being randomly generated, can be shown to be invertible with sufficiently high probability. In the event that a random matrix is generated that is not invertible, one can discard it and wait to receive more shards until an invertible matrix is generated.

III. LIBP2P GOSSIP PROTOCOL: GOSSIPSUB

In practice, OPTIMUMP2P aims to replace the gossip protocol in *libp2p*, a well established and documented peer network stack [27]. Thus, before proceeding with the description of the OPTIMUMP2P algorithm, in this section we examine the details of Gossipsub, the current gossip algorithm that is currently used within libp2p. *libp2p* provides two main gossiping protocols: ***Floodsub*** and ***Gossipsub*** [2], [3].

- ***Floodsub*** is a simple approach where each node forwards every message it receives to all its peers, ensuring broad dissemination with some degree of message redundancy.
- ***Gossipsub*** provides an improvement on Floodsub in the following ways: (i) by organizing messages into topics, with nodes only forwarding messages to peers subscribed to those topics, (ii) by defining a network degree D to limit the number of peers each node exchanges full messages with, and (iii) by utilizing small control messages to optimize the network traffic.

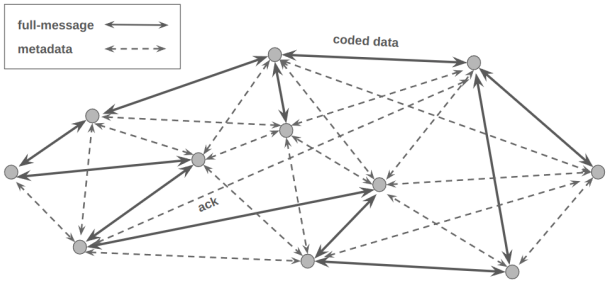

Fig. 1. The figure illustrates two distinct types of peers in the libp2p network: full-message (or mesh) peers and metadata peers

Gossipsub Network Overlay. In Gossipsub, peers establish connections through two types of peerings: ***full-message peerings*** and ***metadata-only peerings***. These two types of connections define the network graph (see Fig. 1). **Full-message** (or *mesh*) peers operate within a sparsely connected network with each peer connected to a degree D other peers, to transmit entire messages across the network. **Metadata** (or *connected*) peers form a densely connected network primarily for exchanging control messages.

The rationale for limiting full-message peerings is to reduce network traffic and increase the available bandwidth. In libp2p's default Gossipsub implementation, $D = 6$ with an acceptable range of 4 to 12. The network degree strikes a balance between several key factors: speed, reliability, resilience, and efficiency. Higher D improves network coverage, and thus message delivery speed, reliability by ensuring messages reach all subscribers, and fault-tolerance by reducing the impact of any peer disconnections. However, increasing the degree raises bandwidth demands and network congestion, as redundant copies of each message are generated.

Control Messages. By controlling the number of full-message peerings, the network can optimize for both performance and resource efficiency. The gossipsub protocol leverages several control messages in order to manage the topology of the gossip network and its peer-to-peer connections (see [28]).

IV. OPTIMUMP2P: THE GALOIS GOSSIP PROTOCOL

In this section we present the OPTIMUMP2P protocol that extends ideas in Gossipsub and is designed to improve the latency achieved by current gossip applications (e.g., a few hundred milliseconds to few seconds). OPTIMUMP2P builds on top of the libp2p communication stack, thus any message published in OPTIMUMP2P, adheres the rules specified by libp2p.

Parameter	Description
$N(v)$	Set of *connected* neighbors of v in $G = (V, E)$
$N_{mesh}(v)$	Set of full-message (*mesh*) neighbors of v in G
$t_{heartbeat}$	The heartbeat time for relaying IHAVE messages
k	Rank of coded-stripes of a message
r	The forwarding threshold
p	Published shard multiplier

TABLE I
Parameters encoded in the protocol for each node v

The OPTIMUMP2P protocol primarily relies on a push-based system, in which the publisher of a message pushes shards to its peers, who then forward combination of received shards to their own peers. The protocol has a fallback for when this system doesn't work; nodes that don't get enough shards to decode a message can request more shards from their peers as necessary. OPTIMUMP2P uses a peering degree D as defined in Gossipsub for defining an overlay network.

At a high level, OPTIMUMP2P works in four stages:

Publisher Propagation: Once a peer v receives a publish($\mathbf{v}$), it first divides the message to be published into k fragments, and encodes those fragments using RLNC into $p * k$ *shards*. It then sends these shards to its full-message neighbors.

Algorithm 1 OPTIMUMP2P Gossip: Data Types, Parameters, State and Signature at node v

Data Types:
$\mathcal{V}$: set of allowed message values
$\mathcal{M} \subseteq H$: set message identifiers

Parameters:
$N(v) \subseteq V$: set of neighbors of $v \in V$ in $G = (V, E)$
$N_{full}(v) \subseteq N(v)$: set of full-message neighbors of v
$t_{heartbeat}$: time interval for garbage collection
$k \in \mathbb{N}$: fragmentation parameter
$r \in \mathbb{N}$: forwarding threshold

State Variables:
msgBuffer $\subseteq \mathcal{M} \times \mathcal{V}$, published message buffer initially $\emptyset$
sendBuffer$[m] \subset N(v)^2 \times \mathbb{Z}^+_{\geq 0} \times (\mathbb{F}_{2^8})^k$, send buffer initially $\emptyset$
shardSet$[m] \subseteq N(v) \times \mathbb{Z}^+_{\geq 0} \times (\mathbb{F}_{2^8})^k$ shards for $m \in \mathcal{M}$ init $\emptyset$
msgDecoded$[m] \in \mathcal{V} \times 2^{\text{shardSet}[m]}$,
↪ decoded value and shards for $m \in \mathcal{M}$ initially $\perp$
isDone$[m] \subseteq N(v)$, set of peers decoded $m \in \mathcal{M}$ init $\emptyset$
doneSent$[m] \subseteq N(v)$,
↪ peers we have sent IDONTWANT for $m \in \mathcal{M}$ init $\emptyset$
iWant$[m] \in N(v)$, peer to send IWANT for $m \in \mathcal{M}$ init $\perp$

Signature:
Input:
publish($\mathbf{v}$), $\mathbf{v} \in \mathcal{V}$
receive-shard(m, s), $m \in \mathcal{M}$, $s \in N(v) \times \mathbb{Z}^+_{\geq 0} \times (\mathbb{F}_{2^8})^k$
receive-done(m, IDONTWANT), $m \in \mathcal{M}$
receive-ihave(m, IHAVE), $m \in \mathcal{M}$
receive-iwant(m, IWANT), $m \in \mathcal{M}$
Output:
deliver($\mathbf{v}$), $\mathbf{v} \in \mathcal{V}$
send-shard(m, s), $m \in \mathcal{M}$, $s \in N(v) \times \mathbb{Z}^+_{\geq 0} \times (\mathbb{F}_{2^8})^k$
send-done(m, IDONTWANT), $m \in \mathcal{M}$
send-ihave(m, IHAVE), $m \in \mathcal{M}$
send-iwant(m, IWANT), $m \in \mathcal{M}$
Internal:
generate-shards()
decode-message(m), $m \in \mathcal{M}$

Shard Processing: When a node v' receives a shard, it adds it to its set of shards for the message. If it has enough shards to decode the message, it does so.

Shard Forwarding: After having processed an incoming shard, a node conditionally forwards it to its own peers. The conditions depend on: (i) whom the shard came from, (ii) how many shards the node has locally, and (iii) the status of the node's peers.

Requesting Additional Shards: Periodically, a peer that does not have enough shards to decode requests more shards from its peers. Peers receiving such request reply with more shards.

A. Optimizations

To further improve performance, OPTIMUMP2P adopts four minor optimizations (color coded in Algorithm 2): (i) publisher flooding (magenta), (ii) forwarding threshold (cyan), (iii) control messages (blue).

Publisher Flooding: The algorithm is designed to aggressively forward shards created by the publisher, since these shards are created early in the message's lifecycle, and always carry new degrees of freedom. Thus, the publisher sends shards to all of its neighbors, i.e. $N(v)$ and not only to mesh peers, i.e. $N_{mesh}(v)$, aiming to expedite the dissemination of the degrees of freedom.

Forwarding Threshold: Each non-publisher node for a message m, maintains a forwarding threshold r and creates and forwards a new shard whenever it collects more than $\frac{r}{k}$ shards in its local set for m. This shard is then sent to the node's peers in $N_{mesh}(v)$. This in contrast to the publisher's policy, as we aim to reduce unnecessary propagation of shards that are unlikely to carry new degrees of freedom to a node's peers.

Control Messages: We use control message similar to the Gossipsub protocol to suppress unnecessary message transmissions and facilitate dissemination of shards to isolated nodes in case of network partitions. The control messages used are: IDONTWANT, IHAVE, IWANT.

B. RLNC External Library

We assume an external library that handles the RLNC encoding, recoding, and decoding process and offers the following interface:

RLNCencode($\mathbf{v}, n$)**:** The RLNCencode operation accepts a data value $\mathbf{v}$ and uses the RLNC encoding (see Section II) to generate n shards.

RLNCrecode(S)**:** The RLNCencode operation accepts a set of shards S and randomly combines the shards in S to generate a new shard s'. Note that the shards in S may be the result of an RLNCencode or another RLNCrecode operation.

RLNCdecode(S)**:** Last, the RLNCdecode operation given a set S of shards, s.t. $|S| \geq k$, it attempts to use those shards to decode the original value $\mathbf{v}$.

We use those operations in the specification of OPTIMUMP2P without providing their detailed description as this is out of the scope of this work. This library however, is implemented together with OPTIMUMP2P for the needs of the evaluation presented in Section V.

C. IOA Specification

The algorithm is formally specified using the IOA notation [33] through group-defined transitions, each characterized by a specific precondition and its effect. We assume that each node operates in a single-threaded mode, executing transitions atomically once their preconditions are met. The execution of these transitions occurs asynchronously. We adopt a *fairness* assumption in the protocol's execution, which suggests that if preconditions are continually met, each node will have infinite opportunities to execute its transitions that satisfy these preconditions. Algorithm 1 presents the data types, the static parameters, and the state variables used, along with the signature of OPTIMUMP2P. Algorithm 2 presents the transitions of all the actions in OPTIMUMP2P.

State Variables. Every node v maintains the following state variables:

Algorithm 2 OPTIMUMP2P Gossip: Transitions at any node $v \in V$ of graph $G = (V, E)$.

Transitions:

// message publishing
Input publish($\mathbf{v}$)$_v$
  Effect:
    msgBuffer $\leftarrow$ msgBuffer $\cup \{\langle \mathcal{H}(\mathbf{v}), \mathbf{v}\rangle\}$

// message delivery
Output deliver($\langle m, \mathbf{v}\rangle$)$_v$
  Precondition:
    msgDecoded$[m] \neq \perp$
  Effect:
    $\langle \mathbf{v}_{dec}, *\rangle \leftarrow$ msgDecoded$[m]$
    $\langle m, \mathbf{v}\rangle \leftarrow \langle m, \mathbf{v}_{dec}\rangle$

// Generate and encode data
Internal generate-shards()$_v$
  Precondition:
    $\langle m, \mathbf{v}\rangle \in$ msgBuffer
  Effect:
    $S \leftarrow$ RLNCencode($\mathbf{v}, k * |N_{mesh}(v)|$)
    shardSet$[m] \leftarrow \{\langle v, s, c_s\rangle : \langle s, c_s\rangle \in S\}$
    sendBuffer$[m] \leftarrow \{\langle v', s'\rangle : v' \in N(v) \wedge s' \in$ shardSet$[m]\}$

// If v' is not done yet send encoded data to v'
Output send-shard($\langle m', \langle p', s', c'_s\rangle\rangle$)$_{v,v'}$
  Precondition:
    $\langle v', \langle p, s, c_s\rangle\rangle \in$ sendBuffer[m]
    $v' \notin$ isDone$[m]$ *// do not send to peer decoded m*
    $v' \neq p$ *// do not send to the publisher*
  Effect:
    $\langle m', \langle p', s', c'_s\rangle\rangle \leftarrow \langle m, \langle p, s, c_s\rangle\rangle$
    sendBuffer$[m] \leftarrow$ sendBuffer$[m] \setminus \{\langle v', \langle p, s, c_s\rangle\rangle\}$

// Receive coded shards from v'
Input receive-shard($\langle m, \langle p, s, c_s\rangle\rangle$)$_{v',v}$
  Effect:
    if $msgDecoded[m] = \perp$ **then**
      shardSet$[m] \leftarrow$ shardSet$[m] \cup \{\langle p, s, c_s\rangle\}$
      if $|$shardSet$[m]| > r/k$ **then**
        $\langle s', c_{s'}\rangle \leftarrow$ RLNCrecode(shardSet[m])
        $B \leftarrow \{\langle v', \langle p, s', c_{s'}\rangle\rangle : v' \in N_{mesh}(v) \setminus \{p\}\}$
        sendBuffer$[m] \leftarrow$ sendBuffer$[m] \cup B$

// check whether a message is decodable
Internal decode-msg(m)$_v$
  Precondition:
    $\exists S \subseteq$ shardSet$[m]$ s.t. $|S| = k$
    $\mathbf{v}_{dec} \leftarrow$ RLNCdecode(S)
    $m = \mathcal{H}(\mathbf{v}_{dec})$
  Effect:
    msgDecoded$[m] \leftarrow \langle \mathbf{v}_{dec}, S\rangle$

// when we decode, send done to mesh neighbors
Output send-done($\langle m,$ IDONTWANT$\rangle$)$_{v,v'}$
  Precondition:
    msgDecoded$[m] \neq \perp$
    $v' \in N_{mesh}(v)$
    $v' \notin$ doneSent$[m]$
  Effect:
    doneSent$[m] \leftarrow$ doneSent$[m] \cup \{v'\}$

// Receive done message from v'
Input receive-done($\langle m,$ IDONTWANT$\rangle$)$_{v',v}$
  Effect:
    isDone$[m] \leftarrow$ isDone$[m] \cup \{v'\}$

// when we decode, send ihave to mesh neighbors
Output send-ihave($\langle m,$ IHAVE$\rangle$)$_{v,v'}$
  Precondition:
    $time.Now() - heartbeat > t_{heartbeat}$
    msgDecoded$[m] \neq \perp$
    $v' \in N(v)$
    $v' \notin$ isDone$[m]$
  Effect:
    $heartbeat \leftarrow time.Now()$

// Receive ihave message from v'
Input receive-ihave($\langle m,$ IHAVE$\rangle$)$_{v',v}$
  Effect:
    isDone$[m] \leftarrow$ isDone$[m] \cup \{v'\}$
    // iwant message if we have not decoded m
    if msgDecoded$[m] = \perp$ **then**
      $iwant[m] \leftarrow v'$

// send iwant to v'
Output send-iwant($\langle m',$ IWANT$\rangle$)$_{v,v'}$
  Precondition:
    $iwant[m] = v'$
  Effect:
    $m' \leftarrow m$
    $iwant[m] \leftarrow \perp$

// Receive iwant message from v'
Input receive-iwant($\langle m,$ IWANT$\rangle$)$_{v',v}$
  Effect:
    // if we have already decoded m
    if msgDecoded$[m] \neq \perp$ **then**
      $\langle *, S\rangle \leftarrow$ msgDecoded$[m]$
      $\langle s', c_{s'}\rangle \leftarrow$ RLNCrecode(S)
      sendBuffer$[m] \leftarrow$ sendBuffer$[m] \cup \{\langle v', \langle p, s', c_{s'}\rangle\rangle\}$

msgBuffer: a temporary buffer that keeps pairs of message ids and message values that have been requested for publishing. The message id is the hash of the value.
shardSet: a key-value map where the keys are message ids and the values are sets of shards for the same message.
msgDecoded: a key-value map where the keys are message ids and the values are pairs of decoded values with a set of shards that were used during the decoding.
doneSent: is a key-value map where the keys are message ids and the value is a set of peers to which we have sent the IDONTWANT message.
isDone: is a key-value map where the keys are message ids and the value is a set of peers from which we have received the IAMDONE message.
iWant: is a key-value map where the keys are message ids and the value is the id of a peer to send the IWANT message.

Transitions. We now describe the actions of the protocol. Note that input actions are always enabled and are triggered by the environment. Both output and internal actions are executed only if their preconditions are satisfied.

publish**:** a node invokes a publish operation when it executes this action. The publish action accepts a value $\mathbf{v}$ to be published and generates the message identifier m by hashing the given value. It then inserts $\langle m, \mathbf{v}\rangle$ into a message buffer (msgBuffer) for further processing.

deliver**:** once a message is decoded the deliver action returns the value of a message to the caller.

generate-shards**:** this action encodes pending messages to be published in msgBuffer into $p * k$ coded shrards using RLNCencode. These shards are then stored in the shardSet, and a tuple $\langle v', s'\rangle$ is added in the sendBuffer for every peer

v' of the publisher and every shard s' generated.

send-shard**:** is executed to send a shard for a message m to a peer v' when an entry $\langle v', * \rangle \in \mathsf{sendBuffer}[m]$, v' has not yet informed v that has decoded the message m, and v' is not the publisher of m. Once the message is sent is removed from the sendBuffer.

receive-shard**:** when peer v receives a shard from v' for a message m, it adds the shard in shardSet if m is not yet decoded; otherwise it discards the shard. Whenever it adds the new shard in its shardSet and that contains more than $fracrk$ shards (see optimization (ii), v generates a new shard by *recoding* the shards received using the RLNCrecode action. It then prepares to send the shard generated to all its full peers by adding the appropriate entries in the sendBuffer. Note that this action is a key to the performance boost of OPTIMUMP2P.

decode-msg**:** if a node v collects more than k shards in shardSet for a message m, then this action is trigger to decode the message m and store the outcome along with the shards used for the decoding in the msgDecoded$[m]$ variable.

send-done**:** if a message m is decoded and the receiver $v' \in N_{mesh}(v)$ was not yet informed, i.e. v' does not appear in doneSent$[m]$, v sends the IDONTWANT control message to v'.

receive-done**:** upon receiving of a $\langle m,$ IDONTWANT$\rangle$ message from v', node v adds v' in its isDone$[m]$.

send-ihave**:** when a $heartbeat$ timer expires at v for a decoded message m then v sends to $N(v)$ peers that are not in isDone[m] set, an IHAVE message for m.

receive-ihave**:** upon receiving of a $\langle m,$ IHAVE$\rangle$ message from a peer v', node v adds v' in its iWant$[m]$ variable if it have not yet decode m to request shards from v'.

send-iwant**:** request shards for m from v'.

receive-iwant**:** upon receiving of a $\langle m,$ IWANT$\rangle$ message from v', generate a new shard by recoding the shards used to decode message m. Then queue the new shard to be sent to v'.

V. EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation of OPTIMUMP2P, and its performance comparison to that of Gossipsub. Our experiments include both simulation results and real-world deployments. As a general observation, encoding and decoding of messages up to **10MB** was accomplished in less than **6ms**, adding a negligible overhead in the latency of operations compared to the communication in the network. Consequently, in the sections that follow, we primarily focus on network metrics, with the times presented being inclusive of both network and computation overheads.

A. OPTIMUMP2P *Simulation Results*

We used the Ethereum tool Ethshadow[2] to simulate gossip in an Ethereum-like network. We built off the work of prior Ethereum research [15], running the same simulations as them with 1,000 nodes, 20% of which have incoming/outgoing bandwidth of 1Gbps/1Gbps and 80% of which have 50Mbps/50Mbps. The publisher always has 1Gbps/1Gbps in order to get consistent simulation results. The latencies between pairs of nodes are based on real-world geographic locations. We first ran a simple experiment, in which a single publisher publishes a single message. We varied message sizes from 128KB to 4096 KB[3], and observed significantly faster arrival times for all message sizes, as shown below. We remark that our observed performance of Gossipsub matches the results in [15], which give us confidence in our reproducibility.

We also ran simulations in which a single publisher published multiple messages (up to 64), with each message having a size of 128 KB. The results appear in Figure 3. Once again, we observed notably faster arrival times in all cases.

B. OPTIMUMP2P *Real-World Experiments*

We performed side-by-side A/B testing of Gossipsub and OPTIMUMP2P, each deployed across 36 geographically distributed identical nodes in Google Cloud Platform (GCP) data centers (Figure 4), roughly mirroring the distribution of Ethereum validator nodes. In each test, nodes propagated large data blobs, simulating transaction blocks. A randomly selected node initiated each gossip round, and propagation was deemed successful once at least 95% of nodes received the message. We varied two main parameters: (i) the message size (from 4MB up to 10MB blobs), and (ii) the publish rate (ranging from isolated single-block sends to rapid bursts up to several messages per second). Both protocols were pushed to carry **100 messages** per run in some high-load scenarios to observe behavior under message bursts.

The performance metrics recorded include the *propagation latency*, i.e, the time for 95% of the nodes to receive and reconstruct the message, and the *delivery ratio*, i.e. fraction of nodes that obtained the message within a fixed time-bound. We also tracked the average per-message delay and its variance to gauge stability.

Propagation Latency and Scalability: Figure 5 presents the average end-to-end propagation delay for 10MB messages under increasing publish rates, comparing RLNC-based OPTIMUMP2P and Gossipsub. Lower bars indicate faster delivery.

At **1 msg/s**, both protocols achieve sub-second latencies; however, Gossipsub exhibits a slightly higher delay (1.0s) than OPTIMUMP2P (0.8s) due to the inefficiencies inherent in gossip-based redundancy and bandwidth usage.

At **10 msg/s**, Gossipsub's delay increases to 2.5s, while OPTIMUMP2P remains consistently lower at 1.5s. Under the highest rate of 20 msg/s, Gossipsub's latency sharply degrades to 4.0s, signaling network saturation and queuing delays. In contrast, OPTIMUMP2P maintains delivery within 1.8–2.0s.

These results demonstrate that **RLNC enables superior scalability** in terms of propagation latency. The coded approach makes more efficient use of network capacity, avoiding redundant transmissions, whereas Gossipsub's performance significantly declines in situations with bursty, high-throughput demands.

[2]https://ethereum.github.io/ethshadow/

[3]These block sizes match the ones found in popular blockchain implementations like Ethereum [1], and Solana [37].

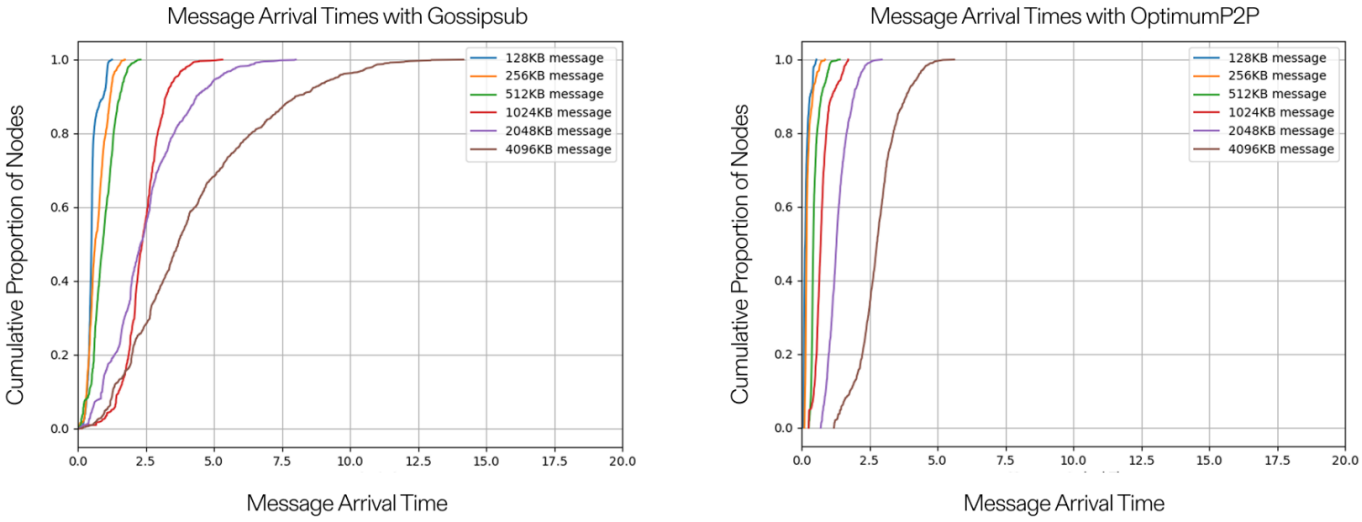

Fig. 2. Comparison of latency between OptimumP2P and Gossipsub by message size.

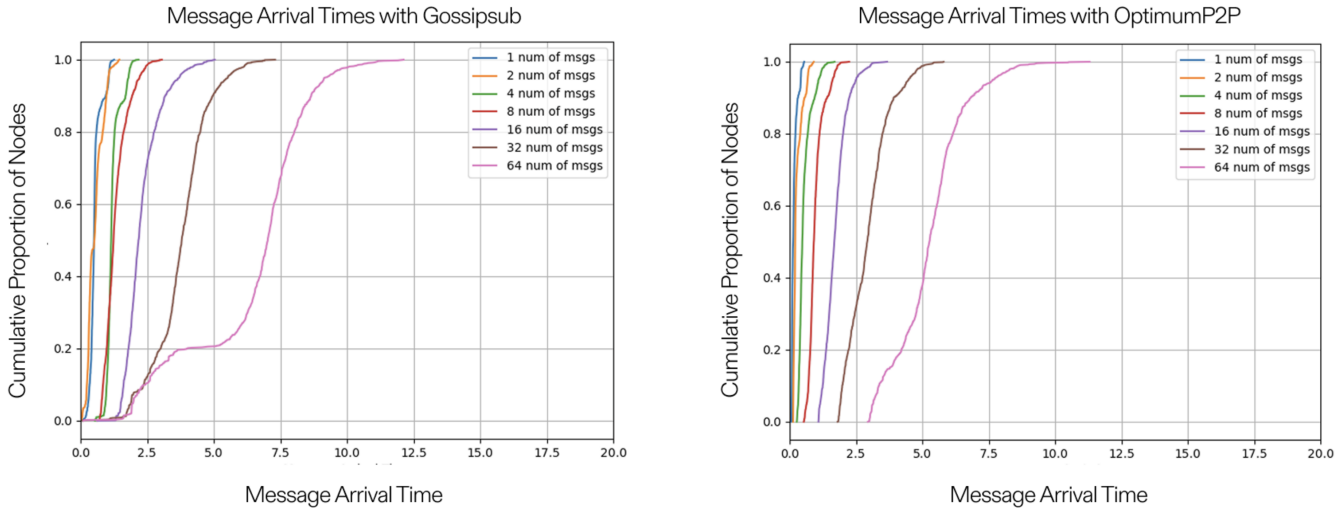

Fig. 3. Comparison of latency between OptimumP2P and Gossipsub by publish rate.

We further evaluated scalability under increasing message sizes at a fixed publish rate of **1 msg/s** testing both 5MB and 10MB payloads. For 5MB blocks, OPTIMUMP2P achieved 100% delivery with an average latency of 1116ms (std = 262ms). Gossipsub delivered 99/100, with a delay of 2293ms and higher variance (std = 712ms). For 10MB blocks, Gossipsub's performance deteriorated sharply—only 84/100 messages were delivered, with an average delay of 15.6s and significant variability (std = 7.3s). OPTIMUMP2P again delivered 100%, with latency held to 1302ms and low deviation (std = 270ms).

These results demonstrate that OPTIMUMP2P remains strong and efficient as message sizes grow, consistently achieving latencies. In contrast, Gossipsub becomes unreliable under larger payloads, succumbing to congestion and protocol overhead.

Throughput and Delivery Success Rate: Figure 6 presents the delivery success rate defined as the percentage of messages delivered network wide under increasing publish rates. The results show a consistent advantage for **RLNC-based OptimumP2P** over **Gossipsub**, especially under high-throughput conditions.

At **1 msg/s**, Gossipsub and OPTIMUMP2P both deliver nearly 100% of messages, and no significant reliability issues are observed at this baseline rate. At **10 msg/s**, Gossipsub drops to **approx. 95%** delivery, likely due to packet loss or transient overload in the gossip mesh, while OPTIMUMP2P maintains a **100%** delivery rate, benefiting from RLNC's redundancy elimination and network coding. At **20 msg/s**, Gossipsub falls to **approx. 80%** delivery, with approximately **1 in 5 messages lost**, likely due to congestion, buffer overflows, or gossip suppression. OPTIMUMP2P continues

Fig. 4. Geographic distribution of the 36 nodes used in each protocol.

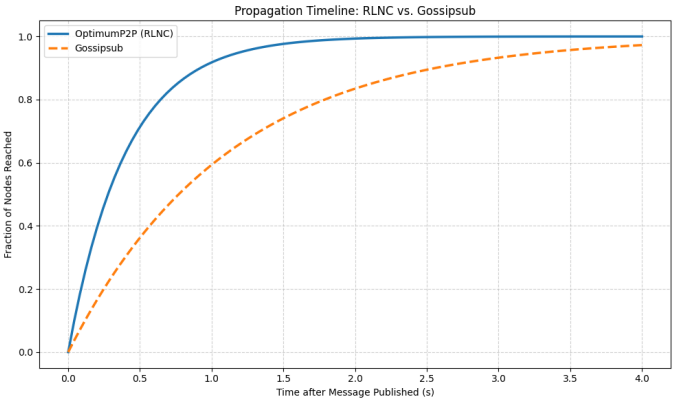

Fig. 7. Propagation Stability of the protocols

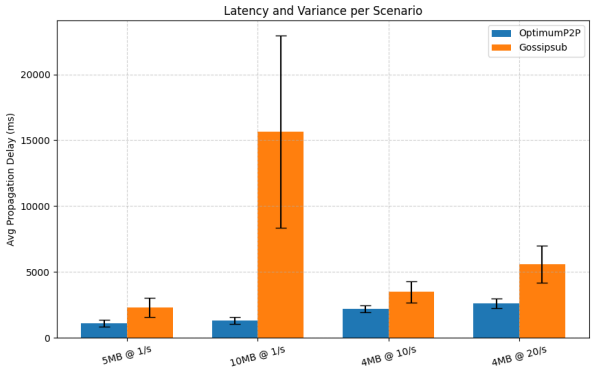

Fig. 5. Latency and Variance at Varying Publishing Rates

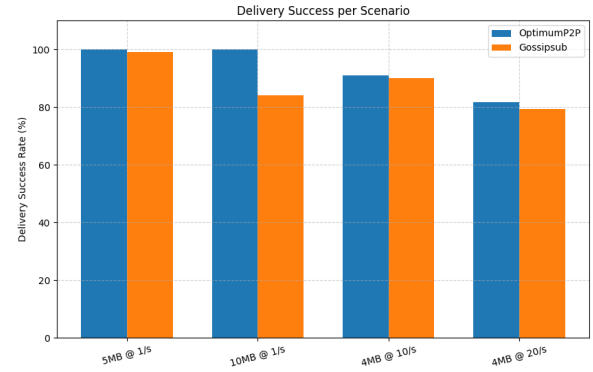

Fig. 6. Delivery Success Rate vs Publishing Rates and Message Sizes

to deliver **99–100%** of messages across the network, with minimal loss despite the high throughput.

These findings underscore the throughput advantage of RLNC-based dissemination. OPTIMUMP2P's ability to deliver coded fragments and recover full messages from partial data ensures robust delivery even under stress. In contrast, Gossipsub's reliance on full-message relays makes it vulnerable to packet drops and network saturation during bursts.

Propagation Stability and Delay Variance: Figure 7 illustrates the **mean propagation delay** and **standard deviation** across nodes for different publish rates. These error bars reveal stable or erratic delivery times under varying network loads.

Across all rates, OPTIMUMP2P exhibits **consistently low variance**, with standard deviation tightly bounded around **±0.2–0.3 seconds**. This suggests a highly predictable propagation process, where most nodes receive messages within a limited time window.

Gossipsub, in contrast, shows **significantly higher delay variability**, especially at **higher publish rates**. At **20 msg/s**, the standard deviation grows to nearly **±0.9 seconds**, indicating that some nodes receive messages much later than others. This inconsistency stems from Gossipsub's multi-hop gossip structure, which can lead to inconsistent dissemination and redundant retransmissions.

The **lower variance** in OPTIMUMP2P designs is due to its **pipeline-friendly, parallel dissemination** using RLNC. Coded fragments are spread uniformly and decoded incrementally, reducing reliance on any single route or node.

These results demonstrate that OPTIMUMP2P provides faster and more predictable delivery, critical for latency-sensitive applications like block propagation in Blockchain systems like Ethereum. In contrast, Gossipsub's performance becomes erratic under load, undermining its suitability for time-critical scenarios.

VI. CONCLUSIONS

We presented OPTIMUMP2P, a gossip protocol that utilizes Random Linear Network Coding (RLNC) to enhance the speed of information propagation in peer-to-peer (p2p) networks. By leveraging the properties of recoding in RLNC, OPTIMUMP2P outperforms current solutions by achieving faster network coverage and reducing message duplication. This enables OPTIMUMP2P to reach peers in the network faster while preserving network bandwidth. In turn, OPTIMUMP2P provides clear benefits to distributed solutions that require information propagation among a set of network nodes, such as block propagation in modern blockchain solutions. The performance gains are evident from our experimental evaluation where we

compare OPTIMUMP2P with the state-of-the-art Gossipsub implementation, both in simulation and real-world setups.

In our experiments we varied both the rate with which messages as published, as well as the size of the messages being gossiped. In all scenarios, OPTIMUMP2P exhibited better performance compared to Gossipsub in terms of scalability, guaranteed message delivery, and delivery latency. In certain cases (as seen in Fig. 5), OPTIMUMP2P was up to 15-fold times faster than Gossipsub. Furthermore, OPTIMUMP2P guarantees delivery of all the published messages even in cases of large block sizes.

In this work we only consider that peers may crash-fail. There is a rich literature for managing Byzantine pollution with RLNC some of which are [25], [26]. A recent extension of this work [35] proposed a new approach that integrates particularly well with OPTIMUMP2P.

Acknowledgments: The computing resources for the deployment of the experimental work in this paper are supported in part by the Startups Cloud Program by Google.

REFERENCES

[1] Ethereum website. https://ethereum.org/en/. Accessed: February 6, 2025.

[2] libp2p gossipsub specs. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md. Accessed: March 11, 2025.

[3] libp2p pubsub website. https://docs.libp2p.io/concepts/pubsub/overview/. Accessed: March 11, 2025.

[4] ANDROULAKI, E., BARGER, A., BORTNIKOV, V., CACHIN, C., CHRISTIDIS, K., DE CARO, A., ENYEART, D., FERRIS, C., LAVENTMAN, G., MANEVICH, Y., ET AL. Hyperledger fabric: a distributed operating system for permissioned blockchains. *Proceedings of the Thirteenth EuroSys Conference* (2018), 1–15.

[5] BIRMAN, K. P. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review 41*, 5 (2007), 8–13.

[6] BOROKHOVICH, M., AVIN, C., AND LOTKER, Z. Tight bounds for algebraic gossip on graphs. In *2010 IEEE International Symposium on Information Theory* (2010), pp. 1758–1762.

[7] BUCHMAN, E. Tendermint: Byzantine fault tolerance in the age of blockchains. Available at https://tendermint.com/static/docs/tendermint.pdf.

[8] BUTERIN, V. Ethereum: A next-generation smart contract and decentralized application platform. Available at https://ethereum.org/en/whitepaper/.

[9] COHEN, A., HAEUPLER, B., AVIN, C., AND MÉDARD, M. Network coding based information spreading in dynamic networks with correlated data. *IEEE Journal on Selected Areas in Communications 33*, 2 (2015), 213–224.

[10] DEB, S., MEDARD, M., AND CHOUTE, C. On random network coding based information dissemination. In *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.* (2005), pp. 278–282.

[11] DEB, S., MÉDARD, M., AND CHOUTE, C. Algebraic gossip: a network coding approach to optimal multiple rumor mongering. *IEEE Transactions on Information Theory 52*, 6 (2006), 2486–2507.

[12] EUGSTER, P. T., GUERRAOUI, R., KERMARREC, A.-M., AND MASSOULIÉ, L. Epidemic information dissemination in distributed systems. *IEEE Computer 37*, 5 (2004), 60–67.

[13] FIROOZ, M. H., AND ROY, S. Data dissemination in wireless networks with network coding. *IEEE Communications Letters 17*, 5 (2013), 944–947.

[14] FOUNDATION, E. Ethereum 2.0 networking specification. https://github.com/ethereum/consensus-specs, 2023. Accessed: 2023-10-10.

[15] FOUNDATION, E. Faster block/blob propagation in ethereum. https://ethresear.ch/t/faster-block-blob-propagation-in-ethereum/21370/1, 2025. Accessed: 2025-03-10.

[16] GONEN, M., LANGBERG, M., AND SPRINTSON, A. Latency and alphabet size in the context of multicast network coding. *IEEE Transactions on Information Theory 68*, 7 (2022), 4289–4300.

[17] GRAHAM, M. A., GANESH, A., AND PIECHOCKI, R. J. Sparse random linear network coding for low latency allcast. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)* (2019), pp. 560–564.

[18] HAEUPLER, B. Analyzing network coding gossip made easy. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2011), STOC '11, Association for Computing Machinery, p. 293–302.

[19] HAEUPLER, B. Analyzing network coding (gossip) made easy. *J. ACM 63*, 3 (Aug. 2016).

[20] HASSANABADI, B., AND VALAEE, S. Reliable periodic safety message broadcasting in vanets using network coding. *IEEE Trans. on Wireless Communications 13*, 3 (2014), 1284–1297.

[21] HO, T., MÉDARD, M., KÖTTER, R., KARGER, D. R., EFFROS, M., SHI, J., AND LEONG, B. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory 52*, 10 (2006), 4413–4430.

[22] JACKSON, J. R. Jobshop-like queueing systems. *Management Science 10*, 1 (1963), 131–142.

[23] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)* (1997), ACM, pp. 654–663.

[24] KERMARREC, A.-M., AND VAN STEEN, M. Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review 41*, 5 (2007), 2–7.

[25] KIM, M., LIMA, L., ZHAO, F., BARROS, J., MÉDARD, M., KOETTER, R., AND KALKER, T. On counteracting byzantine attacks in network coded peer-to-peer networks. *IEEE Journal on Selected Areas in Communications 28*, 5 (June 2010), 692–702.

[26] KIM, M., MÉDARD, M., AND BARROS, J. Algebraic watchdog: Mitigating misbehavior in wireless network coding. *IEEE Journal on Selected Areas in Communications 29*, 10 (Dec. 2011), 1916–1925.

[27] LABS, P. libp2p: A modular network stack for peer-to-peer applications. https://libp2p.io/, 2023. Accessed: 2023-10-10.

[28] LABS, P. gossipsub: An extensible baseline pubsub protocol. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md, 2024.

[29] LANG, S. *Algebra*, 3 ed. Springer, New York, 2002. This book provides a comprehensive treatment of Galois theory and its applications in algebra.

[30] LI, B., AND NIU, D. Random network coding in peer-to-peer networks: From theory to practice. *Proceedings of the IEEE 99*, 3 (2011), 513–523.

[31] LUBY, M. LT codes. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (2002), pp. 271–280.

[32] LUN, D. S., MEDARD, M., KOETTER, R., AND EFFROS, M. On coding for reliable communication over packet networks. *Physical Communication 1*, 1 (2008), 3–20.

[33] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[34] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

[35] NICOLAOU, N., OBI, O., RAJASEKARAN, A., BERGASOV, A., BEZOBCHUK, A., KONWAR, K. M., MEIER, M., PAIVA, S., SINGH, H. P., VISHWANATH, S. S. S., AND MEDARD, M. Optimump2p: Fast and reliable gossiping in p2p networks, 2025. Arxiv URL: https://arxiv.org/abs/2508.04833.

[36] WANG, G., AND LIN, Z. On the performance of multi-message algebraic gossip algorithms in dynamic random geometric graphs. *IEEE Communications Letters 22*, 3 (2018), 470–473.

[37] YAKOVENKO, A. Solana: A new architecture for a high performance blockchain. https://solana.com/solana-whitepaper.pdf, 2017. Accessed: 2025-09-03.

[38] YANG, L., GILAD, Y., AND ALIZADEH, M. Coded transaction broadcasting for high-throughput blockchains.

[39] ZHANG, X., NEGLIA, G., KUROSE, J., TOWSLEY, D., AND WANG, H. Benefits of network coding for unicast application in disruption-tolerant networks. *IEEE/ACM Transactions on Networking 21*, 5 (2013), 1407–1420.