Probabilistic Edge Multicast Routing for the XRP Network

Vytautas Tumas
SEDAN - SnT
University of Luxembourg
Luxembourg, Luxembourg
vytautas.tumas@uni.lu

Sean Rivera
SEDAN - SnT
University of Luxembourg
Luxembourg, Luxembourg
sean.rivera@uni.lu

Damien Magoni
LaBRI - CNRS
University of Bordeaux
Talence, France
magoni@labri.fr

Radu State
SEDAN - SnT
University of Luxembourg
Luxembourg, Luxembourg
radu.state@uni.lu

***Abstract*—The XRP Ledger relies on a trusted set of validator nodes to advance the ledger history. Nodes use flood-based broadcasting to disseminate messages. Flooding offers strong message delivery guarantees at the cost of high network utilisation caused by duplicate messages.**

In this paper, we present *pemcast*, an application layer algorithm for efficient one-to-many message routing. The algorithm leverages limited topology awareness and application layer multicasting to deliver messages in the network. The evaluation shows that compared to flooding and gossiping algorithms, *pemcast* can maintain similar reliability whilst generating significantly less redundant traffic.

***Index Terms*—Blockchain, broadcast, multicast, routing, XRP.**

I. Introduction

XRP Ledger is one of the oldest well-established cryptocurrencies. XRP, its native token, ranked sixth by market capitalisation, in August 2022. Unlike Proof of Work protocols used in Bitcoin or Ethereum, the XRP Ledger relies on trust-based validation to advance the ledger history. As a result, it can handle up to 1,500 transactions per second [1].

The XRP network consists of independent servers that fall into two categories: *validator* or *tracking* servers. Tracking servers process candidate transactions from clients and propagate them through the network. Validator servers perform tracking server tasks and also work with other validators to advance the ledger version.

The servers use a broadcast algorithm to disseminate messages. When a server receives a new message, it forwards (by flooding) it to all nodes except the sender. Flood-based protocols are reliable as they explore every path in the network. However, they generate numerous duplicate messages. Naumenko *et al.* [2] reveal that up to 44% of Bitcoin network traffic is redundant.

Gossip protocols reduce the amount of redundant traffic generated while propagating messages in the network. The term "gossip protocol" was coined by Alan Demers in 1987 [3], who was studying methods to propagate information in unreliable networks. Although gossip protocols were extensively studied at the beginning of the century, they received renewed interest due to the advances in blockchain technology [4], [5].

978-1-6654-3540-6/22/$31.00

In a typical gossip-based algorithm, a node sends a message to a random subset of its 1-hop neighbours. The size of the set is often called a *fanout*. Leitao *et al.* [6] showed a trade-off between the fanout and protocol reliability and an inverse correlation between fanout and message redundancy.

In this paper, we propose a novel probabilistic multicast *application layer* routing algorithm - *pemcast*. It relies on partial network views (neighbourhoods) and multicasting to distribute messages with less redundant traffic. Our key contributions are as follows:

- We propose a novel application layer routing algorithm which uses limited topology awareness and application layer multicasting to achieve efficient message routing.
- We designed distinct metrics to evaluate the performance of *pemcast*, flooding and probabilistic broadcast algorithms.
- We conduct an experimental evaluation using the existing XRP network topology. We demonstrate that *pemcast* in comparison to flooding and gossip-based protocols, can: (1) propagate a message to multiple targets whilst generating fewer duplicates, (2) send the message to fewer nodes not involved in the broadcasting of a message to a given destination.

We arrange the remainder of this paper as follows. In Section II, we discuss existing probabilistic broadcasting solutions. In Section III, we describe the algorithm and discuss the methods to reduce network flooding. In Section IV, we evaluate *pemcast*, and discuss the simulation results. We conclude and discuss future work in Section V.

II. Related Work

Bimodal Multicast [7] was one of the pioneering works to combine multicasting and gossiping to disseminate messages in large-scale distributed systems.

The algorithm has two stages: (1) the message is sent using IP-Multicast, and (2) participants engage in gossip routing to deliver messages lost in the first stage. This approach has well-known drawbacks. IP-Multicast is not widely deployed [8]. Furthermore, the algorithm uses two different protocols, which introduces unnecessary complexity.

Lightweight Probabilistic Broadcast [9] (*lpbcast*), is a decentralised probabilistic broadcast algorithm. Nodes maintain

a local view of a *fixed* size whose members they obtained randomly. When every node in the network is known by multiple other nodes, fault tolerance is preserved.

Scribe [10] is an application layer multicast infrastructure built on top of a Pastry [11] overlay network. Scribe uses multicast groups with multiple senders. Scribe constructs a distribution tree for each group. In the tree, some nodes serve as rendezvous points and root of the multicast tree. Non-root nodes are aware of their parent node and actively monitor the health of that node. Such an approach requires regular heartbeat messages from the root node to notify its children that it is still alive, which generates a high number of control messages when they are not piggy-backed on data messages.

GossipSub [5] is a publish-subscribe messaging system. It constructs an overlay network (*mesh*) per topic. Messages are broadcasted in the mesh. Furthermore, nodes gossip information about the messages they have to nodes not part of the *mesh*. The actual messages are cached so that nodes receiving the gossip can request the messages with a control request.

Distributing messages via routed spanning trees is another popular method. [12]–[14]. These algorithms rely on a central node in the network to behave as a root of the spanning tree. However, this introduces a central failure point.

In 2019, Craig *et al.* implemented a forwarding state reduction mechanism for multi-tree multicast by using Bloom filters. However, their proposal only works in Software Defined Networks [15].

In 2021, Newport *et al.* designed an asynchronous algorithm to implement a new gossip strategy in smartphone-based peer-to-peer networks. [16]. It is one of the most recent papers on gossiping. However, it targets periodic communication, which is not the case in the XRP network.

III. Probabilistic Multicast Routing

A. Algorithm Design

1) Goals: We designed *pemcast* with three goals in mind: 1) Minimise the number of duplicate packets created whilst delivering a message. We consider a message duplicate when some node *u* receives it more than once. 2) Reasonable length paths to the destination node lead through a subset of nodes in the network. Our goal is to reduce the portion of the network explored during message propagation. 3) It is inevitable that nodes in the network will fail, and the neighbourhood view will become stale. Messages must be delivered even when there are inactive nodes (as defined below) in the network.

On the one hand, typical multicast protocols based on a shared tree or on source-rooted trees would significantly reduce the amount of traffic. On the other hand, trees have a high convergence time recovering from failures and must hold numerous memory states. In addition, only a single path exists between any two nodes in the tree. A node could easily interfere with traffic to a given destination. Probabilistic multicast ensures that a message travels to the destination across multiple paths. For these reasons, we do not consider tree-based techniques, and we do not evaluate such protocols in Section IV.

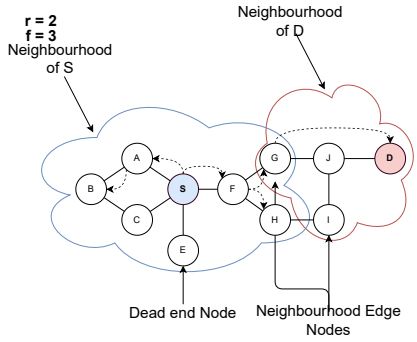


Fig. 1: Example network.

2) Assumptions: We make several assumptions about the behaviour of the nodes. 1) The nodes are not Byzantine faulty. A node can either receive a message and respond to it correctly or not respond at all. 2) The nodes can be either active or inactive. An active node correctly executes the protocol. An inactive node does not respond to received messages. Furthermore, other nodes are not aware of its state.

3) Overlay Network: Formally we define the network in which *pemcast* is running as an unweighted, bi-directional simple graph $G = (V, E)$, where V is a finite set of nodes *u*, and $E \subset V \times V$ a set of links connecting the nodes. A unique, cryptographically secure identifier identifies each node. Peers (i.e., 1-hop neighbours) of *u* are nodes with which *u* has a direct link $P(u) = \{v | \{u, v\} \in E\}$.

Each node in the network maintains a view of all nodes up to *r* hops away. The view is called *neighbourhood* of *u*, and *r* is its radius. neighbourhood of *u* is rooted in *u*, we define it as $N_r(u) = \{v | d_v(u) \leqslant r\}$; *r* is the upper bound on the shortest path length, and $d_v(u)$ is the length of the shortest path between nodes u and v. *neighbourhood edge* is a set of nodes in the neighbourhood that are exactly r hops away or are end-vertices.

We illustrate a neighbourhood in Figure 1[1]. A neighbourhood of node S has a radius of 2. $N_2(S) = \{A, B, C, E, F, G, H\}$, and the edge nodes are $\{B, E, G, H\}$. Note that G belongs to the neighbourhood of both S and D.

4) neighbourhood maintenance: When a new node N' joins the network, N' builds its local neighbourhood. Nodes up to r hops away from N' add it to their neighbourhood. When N' leaves the network, other nodes remove it from their neighbourhoods.

A membership discovery algorithm is responsible for managing this process. The details of these algorithms are outside the scope of this paper. Instead, we defer the reader to previous works [17]–[19].

5) Control Parameters: pemcast is controlled by two parameters *fanout* and *neighbourhood radius*.

- **Fanout** affects the number of neighbourhood edge nodes selected by the sender at each multicasting step. The trade-off is between reliability and redundancy. Higher

[1]For simplicity, we have omitted highlighting neighbourhoods of all other nodes in the network.

values increase the reliability at the cost of increased redundant traffic.

- **neighbourhood radius** controls the maximum length of the shortest path from the root node to every other node in the network. Higher values create larger neighbourhoods, which reduce the amount of traffic. However, large neighbourhoods will have a high membership maintenance cost.

pemcast execution consists of two phases. (1) *Path Discovery* - during which a message is transmitted from an arbitrary source node to some destination node using a combination of multicasting and source routing. (2) *Path Establishment* - the fastest discovered path to the destination is fixed in place. Subsequent messages between the source and destination nodes travel across this path. In the rest of this section, we discuss the phases in more detail.

Algorithm 1: Select edge nodes at node u

```
1  targets ← ∅
2  edgeNodes ← edgeNodes()
3  for |targets| < fanout ∧ !edgeNodes.empty() do
4      node ← edgeNodes.popRandom()
5      path ← neighbourhoodShortestPath(node)
6      if |path| < neighbourhoodRadius then
7          continue
8      if path.contains(m.sender) ∨
9      path.contains(m.source) then
10         continue
11     targets[path[0]].append(path[l_path − 1])
12 return targets
```

B. Path Discovery

During path discovery, a node can have one of the following roles: *Source* (S) - a node from which the message originates. *Destination* (D) - the final recipient of the message. *Sender* - node that multicasts the message to the edge of its neighbourhood. *Receiver* - a node on the neighbourhood edge that receives the message. *Forwarder* - any node simply forwarding the message.

A unique path ID PID_{SD} identifies a path between *source* and *destination* nodes. The path ID uniquely combines the *source* and *destination* node IDs, irrespective whether the message is traveling from S to D or D to S..

Nodes maintain two path tables: *pending paths* and *established paths*. Both tables use the path ID for indexing. The *Pending Paths* table holds candidate paths between a source and destination nodes. For example, the pending paths table for node F for PID_{SD} is $[(S, G), (S, H)]$. The *Established Paths* table holds confirmed paths. For example, $PID_{SD} \rightarrow (S, G)$.

Entries in both tables expire after a period of inactivity. Entries in the established paths table expire when a node does not observe any messages. Similarly, entries in the pending paths expire when the path is not confirmed.

Nodes maintain a view of nodes up to *r* number of hops away. The neighbourhood is stored as a graph. The graph allows nodes to retrieve paths to other nodes in their neighbourhood. In the remainder of this section, we describe node roles in greater detail.

1) Source Node: The role of the source node *S* is to deliver message *m* to the destination node *D*. If a path between S and D exists in the *Established Paths* table, S sends the message over it. Otherwise, *S* initiates the path discovery process.

If S finds at least two paths between itself and D in its neighbourhood, S routes the messages over every found route. Otherwise, S proceeds to multicast the message as follows.

The source node constructs multicast sub-trees for a subset of neighbourhood edge nodes, as depicted in Algorithm 1. First, S selects neighbourhood edge nodes. Next, until a *fanout* number of target nodes are selected, *S* picks a random candidate node and computes the shortest path to it. S skips the candidate node if (1) the path's length is shorter than the neighbourhood radius, implying the node is a dead-end, or (2) the path contains the source or the sender nodes. Otherwise, S adds the candidate to the set of *targets*. A single entry in *targets* is a mapping $(peer, destinations)$, from some peer P of S, to a list of edge nodes that can be reached through P. For example, for node S this could be $(F, [G, H])$, as both G and H nodes are reached via F.

Once S constructs the multicast sub-trees, it sends the messages as depicted in Algorithm 2. For each mapping *m*, the source node prepares a new message, sets its destinations field to $m.destinations$, sends the message to $m.peer$, and finally adds an entry to the list of potential paths.

2) Forwarder Node: A node determines its role in message handling by checking whether its identifier is in the *destinations* field of a message. If the identifier is present, the node behaves as a *receiver*, and *forwarder* otherwise.

The *forwarder* node is responsible for forwarding messages to their respective destinations. For each entry in the message's *destinations* field, the forwarder: (1) computes the shortest neighbourhood path to it, (2) computes a multicast mapping, as outlined in Algorithm 1, and (3) forwards a copy of the message to the next node on the path.

3) Receiver Node: The node behaves as a *receiver* when its identifier is present in the *destinations* field of a message. The *receiver* has two tasks: (1) to unicast the message to the final *destination* node if it is a member of the *forwarder's* neighbourhood. (2) to multicast the message to the edge of its neighbourhood by executing Algorithms 2 and 1

C. Path Establishment

When the destination node D receives a message addressed to itself, it responds with a path acknowledgement, which is sent back to S via the reverse path. Each node on the reverse path executes Algorithm 3 upon receiving a path acknowledgement message. First, a node retrieves the pending path entry from the *Pending Paths* table using the path and sender IDs. The *sender ID* is the identifier of the node that sent the acknowledgement. It identifies which pair of nodes were involved in forwarding the message from the source to the destination. One of the nodes in the pair is always the *Sender ID*. Next, the node forwards the path acknowledgement message to the peer, which is not the sender. For example, node F has a pending path entry (S, G). When F receives a

path acknowledgement from G, it will know to forward the message to S. Finally, the node moves the confirmed entry to the *Established Paths* table and removes it from the *Pending Paths* table.

D. Minimising Flooding

In this section, we discuss how the algorithm design addresses redundant traffic.

1) Effects of the neighbourhood: When the destination node is in the neighbourhood of some node *u*, the node can terminate the multicasting process and proceed to route the message directly to the destination. The neighbourhood's size impacts how soon the message can be routed directly to the destination. Furthermore, multicasting the packet to the edge of the neighbourhood reduces the volume of created traffic. By multicasting, the *forwarder* nodes only propagate the message to selected peers. Therefore, they do not introduce redundant copies of the message to the network. As a result, nodes can use higher fanout values than traditional gossip algorithms without creating duplicate packets.

Algorithm 2: Multicasting path request message m from node u to the edge of $N_n(u)$

```
1 targets ← selectEdgeNodes(m)
2 foreach t ∈ targets do
3     m' ← copy(m)
4     m'.destinations ← t.destinations
5     sendMessage(m')
6     p_ID ← newPathID(m.source, m.destination)
7     addPendingPath(t.peer, p_ID)
```

2) Message Cycling: The *pemcast* algorithm uses pseudo-random *nonce* numbers to uniquely identify messages. Upon receiving a message, a node determines whether it has recently seen a message by consulting its cache. In case of a cache hit, either the message is looping, or the message travelled over multiple paths and is therefore not on the fastest path. In either scenario, the node will drop the message. During path confirmation, cycling is not possible as the message travels on the reverse path. The *nonce* mechanism enables nodes to drop duplicate messages and prevents message cycles without running a control plane algorithm.

In this section, we provided a detailed description of *pemcast*. Next, we discuss the experimental evaluation setup and explore simulation results.

Algorithm 3: Handling of path reply message m in node u

```
1 ID_p ← newPathID(m.source, m.destination)
2 pendPath ← getPendingPath(ID_p, m.sender)
3 next ← pendPath.from = m.sender ?
    pendPath.to : pendPath.from
4 m' ← copy(m)
5 m'.sender ← ID_u
6 m'.target ← next
7 sendMessage(m')
8 Est_u[ID_p] ← pendPath
9 Pnd_u[ID_p] ← ∅
```

IV. Experimental Evaluation

The goal of the experimental evaluation is to determine how well *pemcast* achieves the goals outlined in Section III. In the remainder of this section, we evaluate *pemcast*.

A. Metrics

We used the following metrics to evaluate the performance of the algorithm:

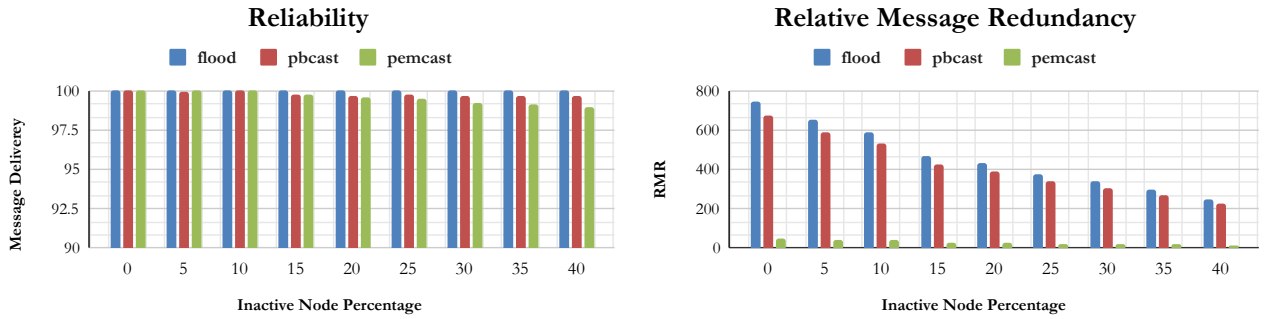
- **Reliability** is the percentage of successful message deliveries. A value of 100% indicates that the protocol delivered every message sent. A flood-based algorithm will achieve a 100% delivery as long as at least one path is available to the destination. Note that we do not consider message loss due to network or transport layer failures.
- **Relative Message Redundancy** [6] (RMR) captures message overheads of a routing protocol. It is expressed as $(m/n-1)-1$, where *m* is the total number of messages sent and *n* is the number of nodes that participated in delivering the message. A zero value indicates that each node sent exactly one message. An increase in value indicates a poorer utilisation of the network. Low RMR may indicate a failure to establish a path. Therefore, it should be followed by a high-reliability value.
- **Path Stretch** is a ratio between the length of the found path and the shortest path. A value of one indicates that an algorithm found the shortest path. Greater values indicate path stretch.
- **Network Explored** is the percentage of nodes that received a message. A zero value suggests that the message did not visit any nodes in the network (and therefore was not sent). A value of 100% indicates that all nodes in the network received the message.

B. Experimental Setup

1) Algorithms: There are many variations of gossip and flooding algorithms, too vast to benchmark against each. Instead, we compare the performance of *pemcast* to that of flood-based broadcast and probabilistic broadcast algorithms as outlined below.

We selected flood-based broadcast as it provides the highest reliability at the expense of efficiency. We implemented the algorithm as follows: After receiving a message, a node forwards it to all of its peers except the sender. The node ignores the message if it has processed it before. The algorithm terminates when the destination node receives the message.

Probabilistic broadcasting (*pbcast*) is a widely used mechanism to distribute messages in a peer-to-peer network. Thus it is vital to compare its performance to *pemcast*. The basic implementation of the probabilistic algorithm is as follows: Each node randomly selects a *fanout* number of peers and forwards the message to them. Nodes repeat this process until the destination node receives the message. There is a high variety [20] of optimisation and peer sampling strategies. We chose the minimalist implementation of the algorithm, which accurately represents the baseline behaviour of the algorithm.



(a) Protocol Reliability

(b) Protocol Relative Message Redundancy

Fig. 2: Protocol performance comparison

2) Simulator: In order to evaluate *pemcast* we implemented a discrete event simulator[2]. The simulator triggers a message propagation event. The event is to deliver a message from a randomly selected, active source node to a randomly selected set of active destination nodes. Multiple destination nodes simulate communication to a group of trusted validator nodes. As of September 2021, the recommended trusted list of validators contained 41 nodes [21]. A single iteration of the simulator is complete when there are no more events to be processed. The simulator iterates until it reaches a 5% Relative Statistical Error in all the metrics outlined previously for a 95% confidence level. We performed the experiments using the XRP Network topology. The network contains 849 nodes, and 8,136 edges and has a mean degree of 19.1, with an SD of 42.

3) Control Parameters: We use a range of different configuration parameters for the experiments. First, to accurately measure the effects of fanout between *pemcast* and *pbcast* we express fanout as a percentage of peers or neighbourhood edge nodes to which to send the message. The fanout ranges from 10% to 95%. Due to the network's density, we use a neighbourhood radius of two for *pemcast*. Finally, up to 40% of nodes in the network were randomly set as inactive.

Due to size limitations, we discuss only the fanout, which achieved the highest reliability. 55% and 90% for *pemcast* and *pbcast* respectively. In the remainder of the section, we compare the performance results of the algorithms in the XRP network.

C. Reliability

We compare the reliability of all three algorithms in Figure 2a, together with relative message redundancy in Figure 2b. The *pemcast* algorithm achieves 100% reliability in a network with up to 10% of inactive nodes. When 40% of nodes are inactive, reliability drops to 99%. *pemcast* achieves this reliability whilst maintaining an RMR between 50 and 12. We measured these numbers with a fanout of 55%. Higher fanout values showed no improvement in reliability.

In contrast, *pbcast* achieves marginally better reliability, between 100% and 99.9% with a fanout of 90%. However, the RMR is significantly higher, between 671 and 224, slightly lower than the flood-based algorithm. Smaller fanout value results in smaller RMR. For example, a fanout of 55% results in RMR between 409 and 132. However, the reliability drops down to between 99% and 93%.

As expected, the flood-based algorithm achieves 100% reliability with up to 40% of inactive nodes. However, the RMR is the highest: 742 in a fully active network and 250 when 40% of nodes are down.

Our proposed routing algorithm achieves a marginally (0.9%) lower reliability than *pbcast* and flooding. However, the amount of generated redundant traffic is significantly lower, *14 times lower than flooding and 13 times lower than pbcast*. The experimental evaluation shows that we have successfully achieved the goals of reducing redundant traffic and maintaining acceptable reliability even when 40% of nodes in the network are down. Next, we compare the overheads of the algorithms.

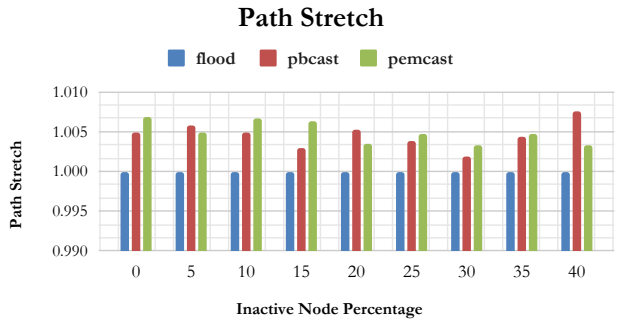
D. Overheads

All algorithms come with overheads, *pemcast* is no exception. Figure 3a depicts the stretch ratio between the established path and the shortest path. We calculate the shortest path using Dijkstra's Algorithm, considering only active nodes. The flood-based algorithm has little impact on the path length. It guarantees to find the shortest path as the algorithm finds all routes to the destination.

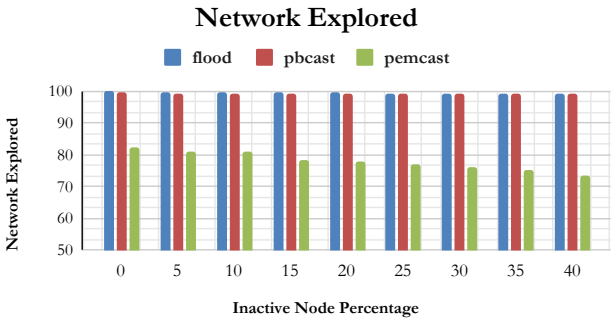
The impact of both *pbcast* and *pemcast* is insignificant. The *pemcast* algorithm achieves this by performing neighbourhood routing. It enables a node to route a message directly to the destination when it is present in the node's neighbourhood. Due to the dense nature of the XRP network, the neighbourhoods are large, with around 500 nodes. Therefore, the probability of sending a message directly to the destination is high.

It is well established [6] that there is a clear trade-off between reliability and network exploration. Intuitively, in the presence of inactive nodes, as we explore more of the network, the higher the probability of finding a path to the destination. We show this in Figure 3b. The figure depicts the percentage of the network explored to achieve the reliability rates discussed in Section IV-C. Messages sent by the flooding algorithm reach

[2]https://bitbucket.org/vytautastumas/pblearn/src/master



(a) Path Stretch Comparison. *As flooding guarantees to find the shortest available path, the path stretch is 1*



(b) Percentage of the Network Explored

Fig. 3: Protocol overhead comparison

100% of nodes in the network. This behaviour is expected. Each node forwards the message to each of its peers (except the sender). Therefore, the message reaches every node in the network. To achieve high-reliability *pbcast* has to use a high fanout value. As a result, messages reach 99% of nodes. In contrast, messages distributed with *pemcast* reach between 82% and 73% of nodes. By multicasting messages to the edge of the neighbourhood, the algorithm can see a higher percentage of the nodes in the network without directly sending messages to them. The experimental results show that we also achieved the second goal of visiting a minimal portion of the network.

V. Conclusions and Future Work

With the advent of blockchain technology, probabilistic broadcasting has been attracting renewed research interest to improve communication efficiency. In this paper, we proposed a novel probabilistic, multicast, *application layer* routing algorithm called *pemcast*. It uses local topology awareness to increase routing efficiency whilst maintaining high reliability. Simulation results show that, compared to flooding and probabilistic broadcasting, *pemcast*: (1) can establish a path whilst sending fewer messages, (2) provides established paths which are closer to the shortest path, in comparison to probabilistic broadcasting, (3) explores a smaller portion of the network. Indeed, *pemcast* requires one order of magnitude fewer messages than the typical broadcasting techniques. A critical piece of our future work is to evaluate the performance of *pemcast* as the communication protocol for the XRP network.

Acknowledgment

This work was financially supported by Ripple UBRI.

References

[1] I. Amores-Sesar, C. Cachin, and J. Mićić, "Security analysis of ripple consensus," *arXiv preprint arXiv:2011.14816*, 2020.

[2] G. Naumenko, G. Maxwell, P. Wuille, A. Fedorova, and I. Beschastnikh, "Erlay: Efficient transaction relay for bitcoin," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, p. 817–831.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *6th ACM Symposium on Principles of Distributed Computing*, 1987.

[4] N. Berendea, H. Mercier, E. Onica, and E. Rivière, "Fair and efficient gossip in hyperledger fabric," *CoRR*, vol. abs/2004.07060, 2020. [Online]. Available: https://arxiv.org/abs/2004.07060

[5] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks," 2020.

[6] J. Leitão, J. Pereira, and L. Rodrigues, *Gossip-Based Broadcast*. Boston, MA: Springer US, 2010, pp. 831–860. [Online]. Available: https://doi.org/10.1007/978-0-387-09751-0_29

[7] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, p. 41–88, May 1999.

[8] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the ip multicast service and architecture," *IEEE Network*, vol. 14, no. 1, pp. 78–88, 2000.

[9] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. KOUZNETSOV Distributed Programming Laboratory, A.-m. Kermarrec, and A.-M. Kermarrec, "Lightweight Probabilistic Broadcast," Tech. Rep. 4, 2003.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," 2001.

[12] G. Malavolta, P. Moreno, A. Kate, and M. Maffei, "Silentwhispers: Enforcing security and privacy in decentralized credit networks," 2017.

[13] S. Roos, M. Beck, and T. Strufe, "Anonymous addresses for efficient and resilient routing in f2f overlays," in *35th IEEE International Conference on Computer Communications*, 2016.

[14] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg, "Settling payments fast and private: Efficient decentralized routing for path-based transactions," 2017.

[15] A. Craig, B. Nandy, and I. Lambadaris, "Forwarding state reduction for multi-tree multicast in software defined networks using bloom filters," in *IEEE International Conference on Communications*, 2019.

[16] C. Newport, A. Weaver, and C. Zheng, "Asynchronous gossip in smartphone peer-to-peer networks," in *17th International Conference on Distributed Computing in Sensor Systems*, 2021.

[17] A. Ganesh, A. Kermarrec, and L. Massoulié, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," in *Networked Group Communication*, 2001.

[18] S. Voulgaris, D. Gavidia, and M. van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *J. Network Syst. Manage.*, vol. 13, pp. 197–217, 06 2005.

[19] J. Leitao, J. Pereira, and L. Rodrigues, "Hyparview: A membership protocol for reliable gossip-based broadcast," in *37th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks*, 2007, pp. 419–429.

[20] D. Gutiérrez-Reina, S. L. T. Marín, P. Johnson, and F. Barrero, "A survey on probabilistic broadcast schemes for wireless ad hoc networks," *Ad Hoc Networks*, vol. 25, pp. 263–292, 2015.

[21] "XRP Validators," Oct 2020, [Online; accessed 21. Sep. 2021]. [Online]. Available: https://xrpcharts.ripple.com/#/validators