École Polytechnique Fédérale de Lausanne

Resilient routing protocol for Dela

Katja Goltsova

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Gaurav Narula
EPFL / DEDIS

Semester Project

Sep 2020 – Jan 2021

Contents

1 Introduction

Dela [2] is a distributed ledger architecture, developed at DEDIS, EPFL. It describes and implements the abstractions that allow a set of nodes agree on a state in a decentralized fashion.

To facilitate the communication between nodes, Dela implements a minimalistic network overlay (mino). Opening a connection between each pair of communicating nodes does not scale, so mino trades latency for reducing the number of open connections, sending a message to the destination via intermediate hops that are other participants of the communication. A routing protocol determines the route of a message by calculating the next hop for a given address. The current tree-based routing protocol minimises the number of open connections between nodes. However, it is not resilient: if a node fails or cannot be reached due to a network failure between the node and its parent, the entire subtree of this node does not receive a message.

Resilient overlay networks have been studied extensively. Andersen et al. [1] introduced the concept in 2001, proposing an overlay network, where nodes improve the quality of service by monitoring the the internet paths among themselves and selecting a good path for a message given the desired metric (e.g. latency, throughput). However, in this overlay each node has a connection with each other, thus limiting scalability. At the other end of the spectrum are peer-to-peer (P2P) overlay networks, which scale to hundreds of thousands of nodes and can handle nodes joining and leaving. Many of these systems (typically those based on Distributed Hash Table) assign a random NodeID to each node and forward messages so that each next hop is closer in the identifier space to the destination. For instance, Pastry [4] and Tapestry [6] employ prefix-based routing: in the typical case, each next hop has a longer common prefix with the destination ID than the previous hop. Chord [5] organizes the nodes in a ring; each node maintains a routing table, i-th entry of which allows it to make a step of size at least 2^i on the ring. The next hop is the farthest node whose id still does not exceed the destination's.

This project designs and implements a resilient routing protocol for Dela. The protocol employs prefix-based routing, inspired by Pastry. With the same average number of hops per message and comparable number of open connections, it improves the resilience of the existing tree-based protocol by $25 - 100\%$, depending on the network condition and the workload.

The rest of this report is organized as follows: section 2 provides a background on the place of the routing protocol in Dela. Then, section 3 describes the protocol design and section 4 discusses the implementation of the protocol within mino. Afterwards, section 5 evaluates the protocol's performance and compares it to the protocol currently used in Dela. Finally, section 6 concludes and discusses future work.

2 Background

2.1 Mino interface

Mino provides two ways to start communication with other nodes: `Call` and `Stream`. `Call` takes a message and a set of addresses and sends the message directly to each node. Opening a direct connection to each address, it does not use the routing protocol. `Stream` takes a set of nodes. This is the way to initiate a longer communication round with this set of nodes.

Nodes can participate in communication after creating a public endpoint: defining a handler for `Call`s and `Stream`s. A response to a `Call` is simply a message. A handler for `Stream` has a receiver to receive messages and a sender to send a response.

A call to `Stream` returns a sender and a receiver and initiates a communication round. The caller is called *orchestrator*. Other nodes' behaviour is defined by the handlers they registered. A typical communication round develops as follows: the orchestrator sends a message to the set of nodes, registered when initiating the round; the nodes' handler is invoked, where the message is

received with the receiver and a reply is sent to some of the participating nodes with the sender. If a follower sends a message to another follower, it is handled by that follower's handler. If a follower sends a message to the orchestrator, it arrives at the receiver, returned by `Stream`.

Dela uses `Stream` for collective signatures and for distributed key generation. Both communication rounds start with the orchestrator broadcasting a message. Collective signature participants simply send a message back to the orchestrator. Distributed key generation participants respond by sending a message to each other participant. We used these two workloads to compare the currently used tree based routing protocol and the protocol from this project.

2.2 Routing interface

Forwarding of packets is determined by a *routing table*. A packet is an abstraction, defined by the routing protocol and encapsulating the source, the destinations and the message. A routing table is created with `GenerateTableFrom(Handshake)` when a node is contacted by another node: the first message sent is guaranteed to be a handshake, which contains sufficient information to construct a routing table. The routing table on the node, booting the protocol, is calculated with `New` based on the participants of the communication round, passed to `Stream`.

Packet routing is handled by `Forward` method of the routing table. It takes a packet and calculates the next hop for it. If a packet has multiple destinations (e.g., if it is the first message of the orchestrator which is broadcast), it is split into several packets, one per next hop. `Forward` signals the inability to route a message to a given destination by marking the destination address as `Void`.

The caller signals that a next hop is unreachable from this node with `OnFailure` method of the routing table. It allows the routing table to recompute an alternative route to destinations, not including this next hop.

When a node contacts another node for the first time, it sends a handshake so that the distant node can initialize its routing table. `PrepareHandshakeFor` method of the initiating node's routing table constructs this handshake.

2.3 Minogrpc

Minogrpc is an implementation of network overlay using gRPC. At the moment, this is the only implementation which uses a routing protocol. Its details, relevant to routing, are described below.

As described above, the node, initiating a communication round, becomes the orchestrator. The orchestrator has two parts: client side and server side, connected with a relay. This relay is the only communication channel between the two sides.

The client side does not participate in the routing protocol. However, the messages have to be sent to the client side to reach the node. Therefore, the only way to route a reply to the orchestrator is to route it to the server side and then use the relay to the client side.

The server side boots the routing protocol: this is the participant whose routing table is created with `New`. It computes the next hops for the orchestrator's broadcast messages and sets up a gRPC relay to each of these nodes. The first message it sends is a handshake, based on which the nodes populate their routing tables. The message keeps propagating among participants, and when it is delivered to all destinations, all nodes participated in forwarding, and therefore have their routing tables initialized.

A node, which initiated a relay, becomes a *parent* of the "passive" node. A node stores one routing table per parent, calculated from the respective handshake. Whenever a message needs to be routed, a node tries to route it using each of the routing tables, until it succeeds.

If a routing table routes a packet to `nil`, this is a special scenario, different from marking the destination as `Void`. In this case, the node sends the message with the relay to the parent.[1].

2.4 Tree routing protocol

The current routing protocol in mino builds a tree topology between participants, minimizing the number of open connections. To guarantee small number of hops per message, the tree height is fixed at 3.

This routing protocol dynamically builds a tree topology, choosing the children of the current node among the nodes that are reachable from it. Once the topology is built, it does not change. This makes the protocol not resilient: a failure of one node or network failure between a node and its parent introduces a partition between the subtree of this node and the rest of the participants.

3 Design

3.1 Goals

This project aims to improve the resilience of the existing routing protocol while keeping the number of open connections small (subquadratic) and routing a message with a logarithmic number of hops. We consider a model where nodes and network can fail, but all the nodes adhere to the protocol.

3.2 Protocol

3.2.1 Basic routing

Each node is assigned a NodeID, which is deterministically computed from the node address. The NodeID space is L-digit numbers with base B.

The routing table is specific for each node. For every prefix of the node's NodeID and every differing next digit, the routing table stores an entry (if available). This entry is the next hop for a node with the corresponding prefix. For example, given Table 1, a message to NodeID 000 will be routed to the node with NodeID 021. Note that for a prefix of size L, the next hop in the routing table is either that node, if there is a node with this NodeID among the participants, or there is no entry.

prefix	next hop
*0**	021
*1**	122
2*1**	-
2*2**	220
20*0*	-
20*2*	202

Table 1: Example routing table for node 201 in 3-digit NodeID space with base 3

We provide an implementation which in addition to the routing table computes *leaf set*: the set of reachable nodes, whose NodeIDs are closest to this node's NodeID. If the destination's

[1]Therefore the only way to reach the client side of the orchestrator is to route a message to the server side and from there to nil

NodeID is in the leaf set, the message is sent directly to destination, bypassing routing with routing table.

3.2.2 Failure handling

If all nodes and network links between nodes are functional, a message is forwarded to a node which shares a longer common prefix with the destination until the message reaches the destination. However, because of failures all next hops with a longer common prefix might become unavailable. In this case we route the message to a node whose NodeID shares with the destination a prefix of the same length as the current node and is numerically closer to the destination's NodeID. Note that in order to avoid loops, it is critical to require the common prefix of the same length, not just being numerically closer.

4 Implementation

4.1 NodeID space

A NodeID is derived from node's address by hashing it into the space of L-digit numbers with base B. L defines the typical number of hops the message takes until it arrives at the destination and is fixed at 5. The base is calculated to achieve a possibly smaller id space without collisions (smaller id space is more dense, which means more redundancy and more resilient routing). By birthday paradox, $B = O(N^{\frac{2}{L}})$, where N is the number of nodes, participating in the communication round.

4.2 Handshake

The handshake contains the information about all the participants of the communication round. Upon receiving the first handshake, a node computes its routing table, which it uses for the entire communication round (adjusting when failures are signaled through `OnFailure`). Further handshakes do not impact the routing table.

We chose to include all participants in the handshake for simplicity. A node can receive the first handshake from any other node, including one that does not share any common prefix with its NodeID, and needs to have sufficient information to serve routing requests in this case.

4.3 Special cases

As discussed in subsection 2.3, to reach the orchestrator, the message has to be routed to the orchestrator's server side. To achieve this, we force the NodeID of the client and server side of the orchestrator to be equal.

The client side of the orchestrator also uses the routing protocol, but is the only player. Similarly to routing from server side to the client side, we route the messages to `nil` so that they are forwarded to the server side and can be routed to destinations.

4.4 Minogrpc changes

The implementation process of this routing protocol allowed us to identify a few points where minogrpc's implementation is tailored to the tree-based protocol, which can be improved. They are outlined below.

When the routing tables routes a message to an address, with which the current node has not opened a relay before, a new relay is open. However, in a non-tree based protocol such node

can be a parent of the current node, i.e. it has contacted the current node before. In this case, we can reuse the relay, already established by the parent.

The orchestrator's client side does not participate in the routing protocol, so the routing tables do not know how to route the messages to it. We patch it by forcing the same NodeID for both client and server, but this should not be decided at the routing protocol level. In particular, the orchestrator's server side should forward the message directly to the client side without invoking the routing protocol.

Currently, a node, participating in a communication round, detects the end of it by monitoring the state of the relays to the parents: if they are all closed, the communication is over. This approach is perfectly correct for tree-based routing protocol, but two issues arise once we start accounting for network failures: first, the relays can be closed due to the network outage, which can be transient. Meanwhile, another node can open a connection to the current node and become its parent. Second, an alternative routing protocol can have loops of established connections, i.e. in minogrpc's case – relays. This does not mean a loop in the routing protocol: for example, node 222 can send all messages to NodeIDs starting with 1 to node 111, while node 111 can send all messages to NodeIDs starting with 2 to node 222. An alternative way to detect the end of a communication round therefore needs to be designed.

4.5 Limitations

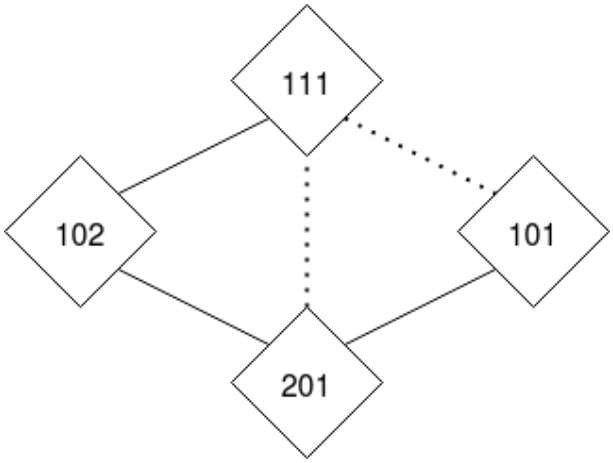

Figure 1: A routing example in face of network failures

Consider Figure 1. Dotted lines stand for links that are down, solid lines – for functional links. Assume that 201's next hop for prefix 1** is 101. A message to 111 will be successfully forwarded to 101, from which it cannot reach the destination. However, a successful routing to the destination would be possible if 101 could signal to 201 that the destination is unreachable from it.

We could improve our choice of next hops by exploiting the information about link latencies. This is not possible at the moment because the communication rounds are short and the information is not preserved across communication rounds.

5 Evaluation

5.1 Setup

We evaluated our implementation of the protocol by simulating message exchange between multiple nodes. We implemented the two scenarios that Dela employs: *replyOrchestrator*, when one node broadcasts a message and others reply to the orchestrator, and *replyAll*, when one node broadcasts a message, and after receiving it all other nodes send a message to all nodes. We compare the two implementations of the routing protocol: with and without the leaf set, and the tree-based routing protocol, currently used in Dela.

To simulate network failures, we used SimNet [3]. SimNet allows the user to control the network topology of a cluster of N nodes, each running the same Docker image. For full control over the network topology, allowing dropping packets, the cluster is deployed on Kubernetes, and each node of the simulated cluster is a Kubernetes pod. We created an autoscaling Kubernetes cluster on Google Cloud Platform with virtual machines with 2 virtual CPUs and 4 GB memory. 5 cluster nodes were enough to run up to 40 simulation nodes (i.e. pods).

The simulation consists of two components: the image that each simulated node is running, and the coordinator program that manages the simulation by executing commands on the nodes and manipulating the network topology, i.e. disconnecting links. The two are detailed below.

By SimNet design, each node in the simulation was running the same image, i.e. could act both as the orchestrator and as a follower. To become an orchestrator, the image had a controller, listening for a command that specified a set of nodes for the communication round and a message to broadcast. To act as a follower, the image had a handler that could process incoming stream requests. Therefore, a communication round started when we executed a command to send a given message to a given set of addresses on one of the nodes. The content of the message determined the scenario that the nodes followed. Both the orchestrator and followers log the received messages.

The simulation starts with a fully functional network. Our coordinator program initiated a communication round by executing a "send message" command on one node. This node became the orchestrator and broadcast the provided message to the provided set of addresses. After this message reached all the nodes, i.e. all nodes had their routing table built, and before the nodes sent the replies, we disconnected a random subset of links[2]. We ran the simulation until all the participants' messages were either delivered or failed to be routed. One simulation round is one communication round with a fixed number of participants that goes according to one of two the described scenarios.

To perform the simulation, we introduced several changes to minogrpc implementation. First, we log open connections, routing failures and packet forwarding. Second, to detect network outages faster, we enable keepalive messages on gRPC relays, sent every 10 seconds (the minimum allowed by gRPC). This way, failed relays are detected in maximum 30 seconds (10 seconds to send a keepalive message and 20 seconds to detect the lack response) instead of several minutes timeout when simply trying to use the relay. Third, we use insecure connection to avoid setting up a certificate between each pair of participants for each simulation round. Fourth, in the current minogrpc implementation, a node detects the end of a communication round as soon as the relays to all its parents close (see subsection 4.4). For the purposes of the simulation, we simply keep the communication round alive until the simulation is over and all resources are freed.

[2]We also implemented disconnecting the links, used by the orchestrator's message, but the negative impact of this approach on the tree routing protocol is much more than on the prefix routing, because in the former, the messages from the participants to the orchestrator use the same links, while for prefix routing this is often not the case

5.2 Routing resilience

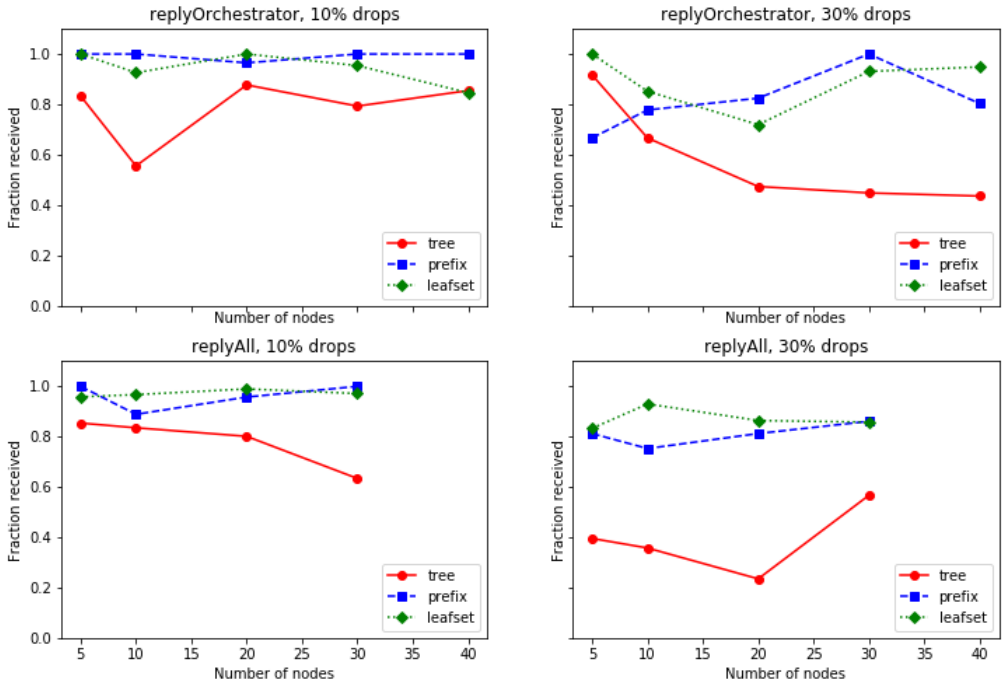

Figure 2: Fraction of delivered messages

Figure 2 presents the analysis of routing protocol resilience for the two communication scenarios and different percentage of failed links. Both implementations of the prefix-based routing significantly outperform the tree-based routing protocol in terms of successfully delivered messages.

5.3 Hop count

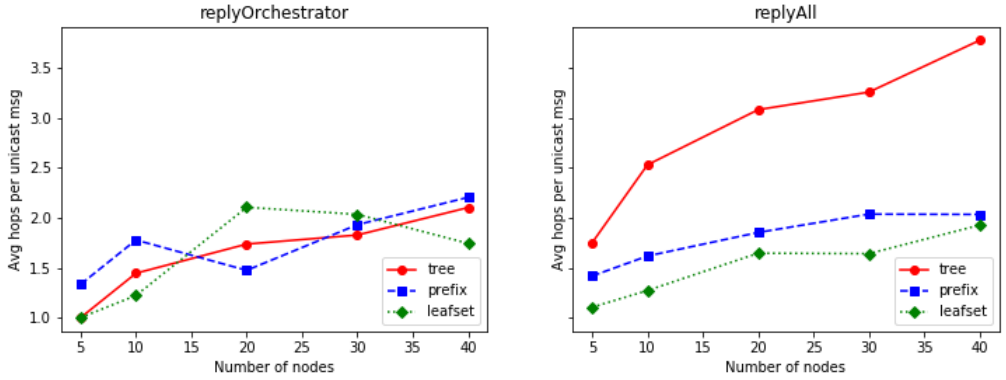

Figure 3: Average number of hops per unicast message, no network failures

Figure 3 shows the average number hops per unicast message with different routing protocols

in a perfectly functioning network. Prefix-based routing protocols perform similarly to the tree-based one. For the tree-based protocol, in the replyAll scenario the average number of hops is approximately doubled because a message to the orchestrator has H hops while a message to another follower has $2H$ hops, where H is the height of the tree. The prefix-based protocol, including leaf set, has a slightly lower average number of hops, because a message is sent directly to a leaf, where in the pure prefix-based protocol it might need to take more hops.

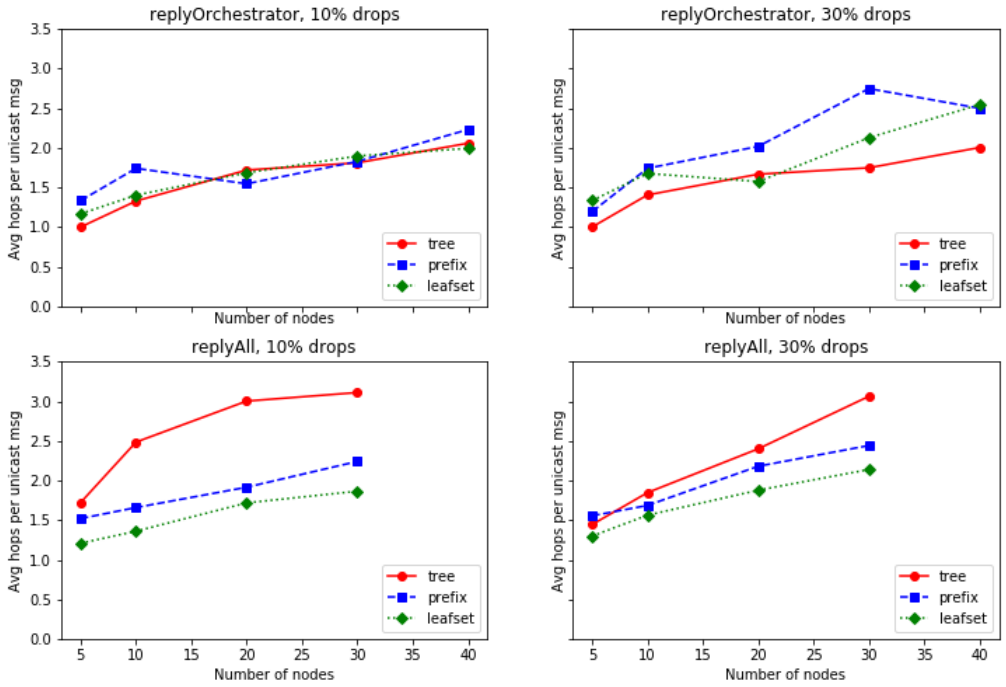

Figure 4: Average number of hops per unicast message

Figure 4 presents the average number of hops per unicast message in a faulty network. The number of hops in the tree-based routing protocol is fixed. The number of hops in prefix-based routing protocols is slightly higher because a longer common prefix is not achieved at every hop.

5.4 Open connections

Figure 5 presents the total number of open connections in a perfectly functioning network. The numbers are the same for tree-based routing protocol, because its topology is always the same. For prefix-based routing protocols, there are significantly more open connections in the replyAll scenario, because more different paths through the network are used.

Figure 6 shows the number of open connections in a faulty network. Only successfully open connections are counted. As expected, prefix-based routing protocols need more connections but the growth is still linear with respect to the number of nodes.

6 Future work

The routing protocol improves on the existing tree-based protocol resilience by exploiting the redundancy in the NodeID space. Further improvements are possible if minogrpc provides a

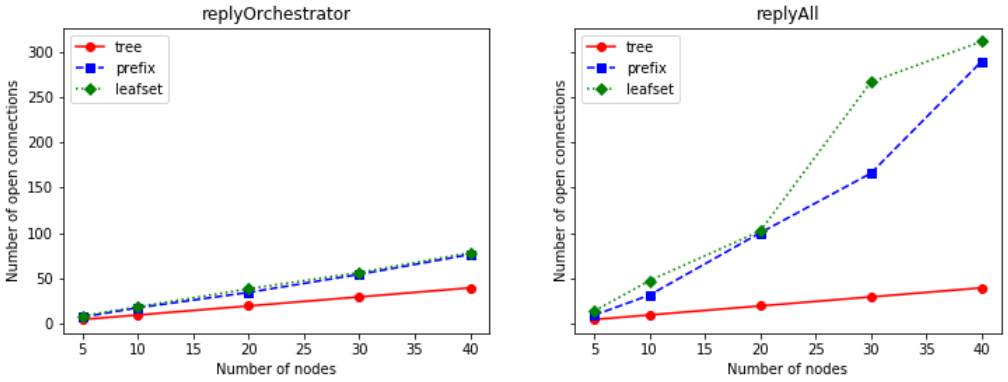

Figure 5: The number of open connections, no network failures

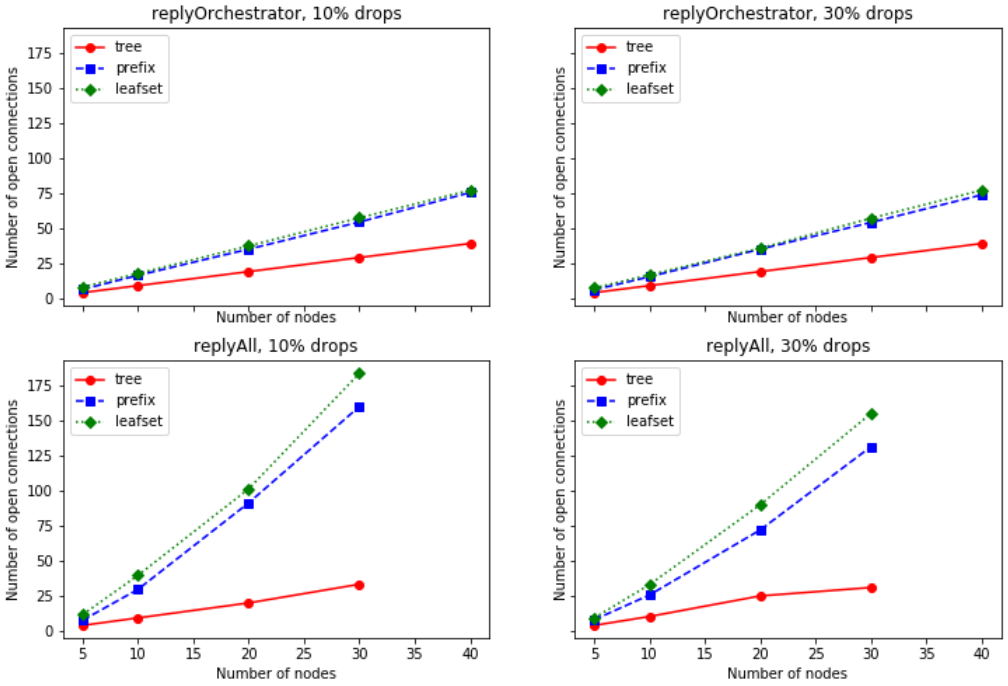

Figure 6: The number of open connections

way to signal last-hop failures (subsection 4.5). Analyzing the data about the network status across different communication rounds might be useful to build latency-aware routing tables.

References

[1] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 131–145, New York, NY, USA, 2001. Association for Computing Machinery.

[2] EPFL/DEDIS. Dela: DEDIS Ledger Architecture. `https://github.com/dedis/dela`, 2016.

[3] EPFL/DEDIS. SimNet. `https://github.com/dedis/simnet`, 2016.

[4] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[5] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, pages 149–160, 2001.

[6] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.