Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Evaluating Gossip Protocols Under Adversarial Conditions: A Simulation-Based Approach

Master's Thesis

Lucas Weitzendorf

lweitzendorf@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Anton Paramonov, Yann Vonlanthen
Prof. Dr. Roger Wattenhofer

February 1, 2026

Abstract

In this thesis, we conduct an in-depth evaluation of two gossip-based message dissemination protocols, GossipSub[1] and Dynamic Optimal Graph[2], using the Shadow Network Simulator[3]. We adapt and significantly expand a peer-to-peer messaging test framework published by lip2p[4] and made these improvements publicly available on GitHub[5]. We then use this implementation to provide insights into the behavior of both protocols in unconstrained, constrained, and adversarial conditions. Further, we demonstrate a simple denial of service attack against the Dynamic Optimal Graph protocol. The obtained results confirm Shadow as a promising tool to understand, benchmark and test novel peer-to-peer protocol designs.

[1] https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md
[2] https://github.com/cometbft/cometbft/blob/main/spec/mempool/gossip/dog.md
[3] https://shadow.github.io
[4] https://github.com/libp2p/test-plans
[5] https://github.com/lweitzendorf/gossip-shadow

Contents

Abstract **i**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 1

1.3 Outline 2

2 Background **3**

2.1 Gossip Protocols 3

2.2 Publish / Subscribe (Pubsub) 3

2.3 GossipSub 4

2.4 Dynamic Optimal Graph (DOG) 5

2.5 Peer Discovery 6

3 Implementation **7**

3.1 Choosing A Simulation Environment 7

3.1.1 ns-3 7

3.1.2 Testground 8

3.1.3 Shadow 8

3.2 Building on Existing Work 9

3.2.1 Framework Structure 9

3.2.2 Shadow Configuration 10

3.2.3 Network Specification 10

3.2.4 Test Harness 12

3.2.5 General Instructions 13

3.2.6 Control Flow Instructions 14

3.2.7 Protocol-Specific Instructions 15

3.3 Major Changes 16

3.3.1 Time Granularity 16
3.3.2 Node-Specific Instructions 17
3.3.3 Event Log Analysis 18
3.3.4 Integrating Dynamic Optimal Graph 19

4 Evaluation **20**
4.1 Baseline Simulations 20
4.1.1 Motivation 20
4.1.2 Setup 21
4.1.3 Results 21
4.1.4 Discussion 27
4.2 Disagreement With Previous Work 27
4.2.1 Motivation 27
4.2.2 Setup 28
4.2.3 Results 28
4.2.4 Discussion 29
4.3 Attacking DOG 30
4.3.1 Motivation 30
4.3.2 Setup 30
4.3.3 Results 31
4.3.4 Discussion 32
4.4 Matching Amplification Factor 33
4.4.1 Motivation 33
4.4.2 Setup 33
4.4.3 Results 34
4.4.4 Discussion 36
4.5 Adaptation to Node Churn 36
4.5.1 Motivation 36
4.5.2 Setup 36
4.5.3 Results 37
4.5.4 Discussion 37
4.6 Two Cliques 38

4.6.1 Motivation 38
4.6.2 Setup 38
4.6.3 Results 38
4.6.4 Discussion 39

5 Conclusion **40**
5.1 Future Work 40

Bibliography **42**

Chapter 1

Introduction

1.1 Motivation

As blockchains such as Ethereum secure increasing amounts of value[1], the robustness of the peer-to-peer networks that underlie them is more important than ever. Many of these networks rely exclusively on gossip-based communication and almost all of them make use of it at some level of the protocol [1]. Even high-performance blockchains like Solana rely on gossip for parts of its information exchange[2]. In this thesis, we set out to test the robustness assumptions that have led to these choices and evaluate the characteristics of two major gossip protocols, **GossipSub**[3] and **Dynamic Optimal Graph**[4], in a simulation environment that can produce repeatable deterministic results.

1.2 Contributions

Our contributions are threefold.

1. We significantly improve on an existing *libp2p* framework for the orchestration, testing and benchmarking of peer-to-peer messaging protocols. This is done by **addressing several scalability concerns**, integrating the DOG protocol and greatly expanding the metrics collected from simulation runs.

2. We demonstrate these improvements by **running large-scale simulations** of both DOG and GossipSub networks and evaluate their performance. In doing so, we contradict previous evaluation work by B. Faivre [2].

3. We show the successful execution of a denial of service attack against the **DOG protocol**.

[1] https://defillama.com/chain/ethereum?stablecoinsMcap=true
[2] https://docs.anza.xyz/validator/gossip
[3] https://github.com/libp2p/specs/tree/master/pubsub/gossipsub
[4] https://github.com/cometbft/cometbft/blob/main/spec/mempool/gossip/dog.md

1.3 Outline

In chapter 2, we give a brief summary of concepts that may be required to understand later parts of the thesis. We walk through details of the evaluated dissemination protocols and other components required for a functional peer-to-peer network.

Chapter 3 summarizes the technical contributions made, explains how both the original framework and our changes to it operate, and lays out other implementation details that might be of interest.

Finally, chapter 4 discusses the results obtained. We step through the conducted experiments one-by-one, explain their motivation and whether the results satisfactorily answer our research questions.

Chapter 2

Background

Understanding the methodology of the experiments conducted within the scope of this thesis requires knowledge of some fundamental building blocks. This chapter introduces the most important protocols and concepts involved.

2.1 Gossip Protocols

Gossiping is a simple dissemination paradigm in which network participants periodically exchange information with their **local peers**. Individual nodes do not have a holistic awareness of global network state or make an effort to coordinate communication beyond their local scope. These can also be referred to as **epidemic protocols**, given that their dissemination patterns resembles the way in which a disease would spread between organisms[3]. Its ease of implementation and high robustness have made gossiping a popular choice in environments that don't impose strong performance constraints.

2.2 Publish / Subscribe (Pubsub)

Publish / subscribe, also referred to as *pubsub*, is one of the major patterns many modern messaging protocol designs fall into. The core idea behind the pubsub architecture is to exchange messages based on shared topics of interest. Network participants **subscribe to topics** they wish to receive messages in without knowing the identity of potential publishers, and publishers send messages to peers based on their subscriptions. For each published message, the protocol will attempt eventual delivery to all subscribers of its topic[1].

[1] https://docs.libp2p.io/concepts/pubsub/overview

Parameter	Purpose	Suggested Default
D	Target mesh peer count	6
D_{low}	Lower bound of mesh peer count	4
D_{high}	Upper bound of mesh peer count	12

Table 2.1: GossipSub Mesh Parameters[2]

2.3 GossipSub

As its name might suggest, GossipSub is a gossip-based pubsub protocol. It has seen major adoption as the main dissemination protocol for both the **Ethereum consensus layer** (ETH 2) and **Filecoin** [1, 4]. For the purposes of this thesis, we are mainly interested in the features of its first two iterations: the **v1.0 base protocol**[2] and **v1.1 security extensions**[3]. GossipSub v1.2[4] adds additional minor improvements that do not significantly affect our experiments. The GossipSub protocol can be seen as a major extension to **floodsub**, a minimal implementation of pubsub in which an incoming message is simply forwarded to all peers subscribed to its topic [1].

The nature of naive floodsub renders it both inefficient and vulnerable to even simple attacks. GossipSub v1.0 improves on this by maintaining two separate layers of communication. By default, connected peers are only used to gossip about **message metadata**. Nodes can send **IHAVE(ids)** control messages to indicate that they received one or more messages with the specified IDs. In response, its peers can reply with **IWANT(ids)** to request the messages be forwarded to them. This is GossipSub's **lazy pull** mechanism [1].

In addition to this, GossipSub maintains a subset of **full-message** peers for each topic, referred to as a **mesh**. Full-message peers are used for **eager push** semantics, in which incoming message payloads are immediately forwarded to peers in the corresponding topic mesh. When a peer is added to the mesh, it will be sent a **GRAFT(topic)** control message to notify it of the change and establish an agreement on the change in communication layer. Conversely, peers removed from the mesh are sent a **PRUNE(topic)** control message. The initial selection of mesh peers is random, with further maintenance being subject to the parameters in 2.1.

Let $d = |mesh[topic]|$ be the number of peers in a given topic mesh. Mesh size d is expected to fluctuate within a range of $[D_{low}, D_{high}]$ with $D_{low} \leq D \leq D_{high}$.

[2] https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md

[3] https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md

[4] https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.2.md

If $d > D_{high}$, the protocol will attempt to prune $D_{high} - D$ peers from the mesh. If $d < D_{low}$, it will try to graft $D - D_{low}$ new peers into the mesh. The choice of D is GossipSub's main mechanism to control **message amplification**. A larger mesh will increase bandwidth consumption as messages are forwarded to more peers, but is expected to reduce propagation delay in most networks as fewer hops are required to reach an equivalent number of nodes [1].

GossipSub v1.1[3] adds additional protections to the protocol. **Peer scoring**[5] replaces the naive random subsampling for full-message peers. In order to achieve this, each node keeps track of a local score for each of its peers based on metrics such as message validity and delivery time. This produces a quantifiable view of peer performance that is intended to allow the timely replacement of unreliable or malicious peers.

2.4 Dynamic Optimal Graph (DOG)

The DOG protocol was developed as a replacement for the initial floodsub implementation of mempool transaction dissemination in **CometBFT**[6]. Its main design goal is to reduce bandwidth usage by targeting **message amplification** directly with a **target redundancy**.

Each node in a Dynamic Optimal Graph network keeps track of its empirical redundancy factor by noting the average message duplication count within a specified interval. Concretely, each message being received exactly n times implies a redundancy factor of $n - 1$. By default, target redundancy is set to 1.0, aiming for each message to be received twice on average[7].

The initial state of DOG is identical to that of floodsub, meaning it will forward messages to all connected peers. The protocol takes note of its redundancy factor at fixed intervals (e.g. 1 second). If the observed redundancy exceeds the specified target redundancy, a **HaveTx(id)** control message will be sent to one of the peers a duplicate was received from. Each node also keeps track of a simple forwarding table, storing which peer a message was initially received from and which peers it was forwarded to. If a given node B receives a **HaveTx** response from node C for a message it received from node A, node B will cease forwarding messages from node A to node C. This process of disabling routes between pairs of nodes can be repeated once at each interval until the target redundancy is reached. The protocol imposes this rate limit on itself to avoid unintentionally disabling routes beyond a functional limit. Given that **HaveTx** messages are issued to peers who sent a previously received message, the protocol design intends

[5] https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md#peer-scoring

[6] https://github.com/cometbft/cometbft

[7] https://github.com/cometbft/cometbft/blob/main/spec/mempool/gossip/dog.md

for low latency connections to be preserved, minimizing the impact on delivery time while reducing bandwidth[7].

In the scenario where a node finds itself receiving too few messages, it can issue a **ResetRoute** control message to one of its peers. The receiving node will respond by reopening a previously disabled route to the peer from whom it received the control message[7].

2.5 Peer Discovery

Although dissemination protocols clearly define interaction and information exchange with connected peers, they rely on an additional discovery layer to identify potential peers to establish a connection with when joining the network. In real-world applications, protocol clients usually encode a static set of **bootstrap nodes** for initial connectivity. From there, primitives such as **distributed hash tables (DHT)** with structured node records may be used to establish additional connections [5]. A specific example of this can be found in Ethereum clients, which rely on an implementation of the **Node Discovery Protocol v5 (discv5)**[8].

In our experiments, the role of the discovery layer is taken over by simulation instructions. Individual nodes are given a concrete list of potential peers at predetermined times in the simulation. Further details of how we implemented node orchestration are laid out in the following chapter.

[8] https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md

Chapter 3

Implementation

The simulation of a network environment for the purposes of testing, debugging, or other experimentation is an established problem. As such, there exist many solutions that aim to solve it at various levels of abstraction. A key component of this thesis was to select a network engine that would underlie our experiments.

3.1 Choosing A Simulation Environment

At their core, the requirements for said engine were the orchestration of a large number of nodes (at least 1000) in a manner that would be applicable to a broad variety of potential dissemination protocol implementations. Our list of candidates quickly narrowed to a list of three.

3.1.1 ns-3

Initially, *ns-3*[1] seemed like a strong contender. It is a discrete-event network simulator, targeted mainly at research use. It allows for fine-grained control of network flows and detailed congestion simulation, down to the packet level. Unfortunately, this level of detail also renders it resource-intensive to run, making it challenging to execute simulations at the scale we would have liked. However, its largest disadvantage for our purposes was how it integrates with programs on top of it. *ns-3* is structured as a C++ library that user programs need to be compiled against, either implemented in C++ themselves or in Python with appropriate bindings, the former being the preferred option [2]. Given that neither C++ nor Python are common implementation languages for peer-to-peer protocols, *ns-3* ended up being too hard to integrate with and we moved on to other options.

[1] https://www.nsnam.org
[2] https://www.nsnam.org/documentation

3.1.2 Testground

Testground[3] is a purpose-built platform for testing and benchmarking of peer-to-peer applications[4]. As such, it is very extensive in its orchestration capabilities and would have offered all the tools needed to perform our experiments. *Testground* has also seen significant uptake from protocol developers for automated testing[5], which would have given us reference integrations to base our work on. The framework was ultimately only let down by two things. First, its repository had not seen a release since March 2023, which made us question whether it was still sensible to start building a new project on top of it in 2025. Second, while it does support many languages and the framework is highly expressive, specifying scenarios within it requires a significant amount of work for each combination of protocol and scenario. Ultimately, despite being feasible to build on top of, there was an even better choice for what we were trying to achieve.

3.1.3 Shadow

Shadow[6], being another discrete-event simulator, is more similar to *ns-3* than *Testground* in many ways. Unlike *ns-3* however, it does not function as a library that needs to be linked into the user program. Shadow is able to run unmodified Linux binaries by intercepting appropriate system calls. This makes it highly flexible, as it allows for free choice of programming language and toolchain for processes deployed in its environment. Further, the use of unmodified binaries opens up the possibility of moving experiments onto real-world machines after trialing them in a simulation environment. *Shadow* does also have a set of limitations, primarily in not being able to emulate all system calls a Linux binary might want to make and possessing a more coarse network simulation model than the likes of *ns-3*. It also offers little orchestration capability directly within the framework that will have to be implemented elsewhere. We felt that these downsides were strongly outweighed by the flexibility and ease of use it offered, and that this would not significantly impact the quality of our experiments, which made *Shadow* our final choice. This choice was further solidified by the discovery that **libp2p** had chosen to migrate its own test suite from *Testground* to *Shadow* in 2023[7].

[3] https://github.com/testground/testground
[4] https://docs.testground.ai
[5] https://github.com/sigp/gossipsub-testground
[6] https://github.com/shadow/shadow
[7] https://github.com/libp2p/test-plans/issues/103

3.2 Building on Existing Work

Having chosen Shadow as the simulation environment, we though it prudent to look for existing integrations of messaging protocols so as not to replicate previous work. Eventually, we came across the **libp2p test-plans** repository[8]. Its **GossipSub interoperability tests**[9], built on Shadow, became a valuable resource for our work. This chapter details the structure of the simulation framework and how we adapted it for our purposes[10].

3.2.1 Framework Structure

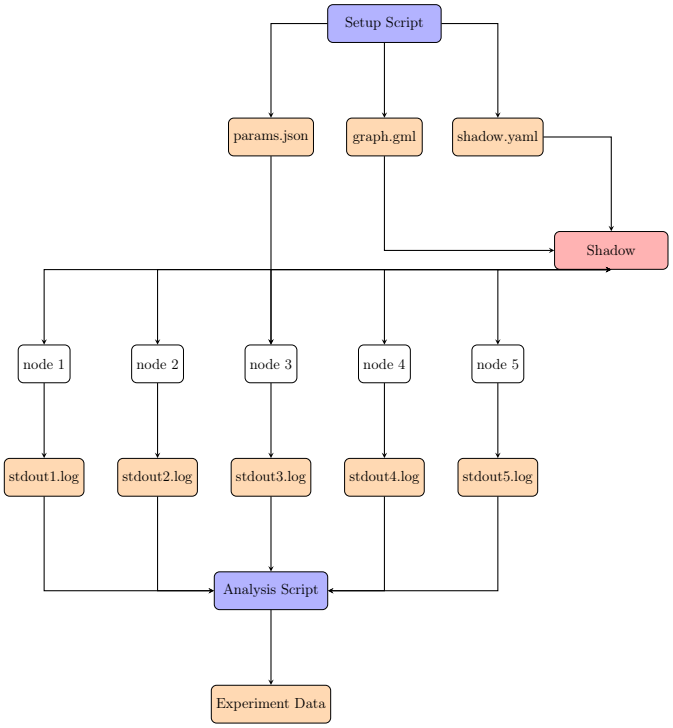


Figure 3.1: Simulation Pipeline

Figure 3.1 provides a high-level overview of the framework's components, artifacts and their interaction with each other. To provide context for further explanation, it should be pointed out that with Shadow, each simulated node is a real process running on the operating system[3]. Simulations require a configuration file (**shadow.yaml**) to instruct Shadow on how to set up these processes[11].

[8] https://github.com/libp2p/test-plans
[9] https://github.com/libp2p/test-plans/tree/master/gossipsub-interop
[10] https://github.com/lweitzendorf/gossip-shadow
[11] https://shadow.github.io/docs/guide/shadow_config_spec.html

Further, we need instructions for individual nodes (**params.json**) in order to control their actions at runtime. Both of these files are generated by the **setup script**.

3.2.2 Shadow Configuration

The **shadow.yaml** file adheres to the general structure shown in 3.1.

Listing 3.1: Shadow Configuration File

```
general:
  model_unblocked_syscall_latency: true
hosts:
  node0:
    network_node_id: 5
    processes:
      args: --params params.json
      path: go-libp2p/gossipsub-bin
  node1:
    network_node_id: 1
    processes:
      args: --params params.json
      path: go-libp2p/gossipsub-bin
  node2:
  ...
network:
  graph:
    file:
      path: graph.gml
    type: gml
```

The **hosts** section provides a specification for each simulated node. As host name, we use the prefix "node" followed by a 0-based index. The **path** points to the compiled binary, which is expected to accept a **params** argument for a file containing the simulation scenario instructions, provided in a JSON file. Each node is further assigned to a **network node**, as defined in the network section of the configuration. In an effort to maintain a manageable size of the main configuration file, the network topology is specified by a separate GML[12] file.

3.2.3 Network Specification

Shadow models its network topology as a graph. The nodes of the network graph allow for the assignment of a **maximum bandwidth** for both upload and

[12] https://www.ogc.org/standards/gml

download, while each edge between these nodes can be assigned a **latency** and **packet loss** percentage. An example of such a graph description is shown in 3.2. Note that the graph can contain self-loops, meaning the communication between two processes on the same network node may have non-zero latency and packet loss.

Listing 3.2: Shadow Network Configuration

```
graph [
  directed 1
  node [
    id 0
    label "australia"
    host_bandwidth_up "50 Mbit"
    host_bandwidth_down "100 Mbit"
  ]
  node [
    id 1
    label "europe"
    host_bandwidth_up "100 Mbit"
    host_bandwidth_down "500 Mbit"
  ]
  ...
  edge [
    source 0
    target 0
    label "australia to australia"
    latency "2 ms"
    packet_loss 0.0
  ]
  edge [
    source 0
    target 2
    label "australia to east_asia"
    latency "110 ms"
    packet_loss 0.0
  ]
  ...
]
```

The base **libp2p** repository[13] contains a static set of locations and associated communication latencies to probabilistically generate network configurations that closely mimic the geographic realities that have been observed on the Ethereum

[13] https://github.com/libp2p/test-plans/blob/master/gossipsub-interop/network_graph.py

Location	Weight
australia	290
europe	5,599
na_east	1,059
na_west	161
east_asia	2,894
west_asia	1,240
south_america	47
south_africa	36

Table 3.1: Location Weights for "Real World" network configuration

source / target	australia	europe	na_east	na_west	east_asia	west_asia	south_america	south_africa
australia	2	165	150	110	110	180	190	220
europe	165	2	70	110	125	60	140	95
na_east	150	70	2	60	140	110	100	130
na_west	110	110	60	2	100	150	100	160
east_asia	110	125	140	100	4	110	175	175
west_asia	180	60	110	150	110	5	145	110
south_america	190	140	100	100	175	145	7	190
south_africa	220	95	130	160	175	110	195	7

Table 3.2: Edge latencies in ms for "Real World" network configuration

mainnet. We will refer to configurations generated from this as "Real World"" network configurations. During setup, the assignment of hosts to locations is done via weighted sampling, with exact weights listed in 3.1. These locations are then connected with latencies as shown in 3.2. Locations are further split into multiple actual network nodes to achieve bandwidth diversity within each given location. The weights for this are listed in 3.3.

3.2.4 Test Harness

Having covered the configuration of Shadow itself, we move on to the node binaries. As mentioned previously, these binaries are expected to accept the path to a JSON file containing simulation instructions in the form of a `--params` argument. This is done via a small layer atop the actual dissemination protocol implementation, which we refer to as the **test harness**. The harness sequentially

Upload Bandwidth (Mbps)	Download Bandwidth (Mbps)	Weight
10,000	10,000	5
1000	250	15
100	20	75
20	5	5

Table 3.3: Bandwidth Weights for "Real World" network configuration

parses the given instructions and interacts with the protocol implementation to execute their respective actions one at a time. Further, it is responsible for effective tracing and the production of corresponding event logs in JSON format for later analysis. See 3.3 for an example of such an event log.

Listing 3.3: Event log for received message

```
{
    "time": "2000-01-01T00:02:00.01600405Z",
    "level": "INFO",
    "msg": "Received␣Message",
    "service": "gossipsub",
    "from":  "12
       D3KooWNRk8VBuTJTYyTbnJC7Nj2UN5jij4dJMo8wtSGT2hRzRP
       ",
    "topic": "topic-a",
    "id": 2
}
```

The framework implements such harnesses for the libp2p GossipSub implementation in both **Go**[14] and **Rust**[15]. As the set of distinct instructions is rather small, and they are an integral part of defining simulation scenarios, we will briefly enumerate them.

3.2.5 General Instructions

WaitUntil

The wait instruction defines a duration in seconds relative to the start of the simulation (**elapsedSeconds**) until which to wait before moving on to the next instruction. In this time, the main dissemination protocol still continues to run in the background and may receive messages, forward them or any other action it does not explicitly need to be instructed to undertake. **WaitUntil** serves as a simple synchronization primitive to coarsely align nodes on the timing of their instructions.

```
{
  "type": "waitUntil",
  "elapsedSeconds": int
}
```

[14] https://github.com/libp2p/test-plans/tree/master/gossipsub-interop/go-libp2p
[15] https://github.com/libp2p/test-plans/tree/master/gossipsub-interop/rust-libp2p

Connect

Connect instructions function as the peer discovery layer for simulated nodes. Each instruction contains a list of node IDs (**connectTo**), for which the harness will subsequently attempt to dynamically obtain IP addresses from Shadow and dial the corresponding peers.

```
{
  "type": "connect",
  "connectTo": List[NodeID]
}
```

Publish

The **Publish** instruction commands a node to spawn a new message to be sent to the rest of the network. It is given a **message ID**, **message size** and **topic ID** to do so.

```
{
  "type": "publish",
  "messageID": int,
  "messageSizeBytes": int,
  "topicID": str
}
```

3.2.6 Control Flow Instructions

As the framework only creates a single instruction file that is read by all nodes, there arises the need for a mechanism to create divergent behavior for specific subsets of nodes. This can be achieved via the **IfNodeIDEquals** instruction. When encountering the instruction, the test harness will compare its own node ID to the provided **nodeID** field. Iff it matches, it will execute the instruction nested within. This construction is most useful in conjunction with **Connect** and **Publish** instructions, individual instances of which are only expected to be executed by a single node.

IfNodeIDEquals

```
{
  "type": "ifNodeIDEquals",
  "nodeID": NodeID,
  "instruction": ScriptInstruction
```

```
}
```

3.2.7 Protocol-Specific Instructions

The framework also contains three instructions that are only applicable to certain protocol designs. **InitGossipSub** explicitly provides a way to specify GossipSub initialization parameters on startup. **SubscribeTopic** and **SetTopicValidationDelay** are slightly broader in their application and can work with any pubsub protocol. The latter provides a delay in seconds to mimic the message processing delay on a real machine. We opted not to use **SetTopicValidationDelay** in our experiments.

InitGossipSub

```
{
  "type": "initGossipSub",
  "gossipSubParams": {
    "D": int,
    "Dlo": int,
    "Dhi": int,
    "Dscore": int,
    "Dout": int,
    "HistoryLength": int,
    "HistoryGossip": int,
    "Dlazy": int,
    "GossipFactor": float,
    "GossipRetransmission": int,
    "HeartbeatInitialDelay": float,
    "HeartbeatInterval": float,
    "SlowHeartbeatWarning": float,
    "FanoutTTL": float,
    "PrunePeers": int,
    "PruneBackoff": float,
    "UnsubscribeBackoff": float,
    "Connectors": int,
    "MaxPendingConnections": int,
    "ConnectionTimeout": float,
    "DirectConnectTicks": int,
    "DirectConnectInitialDelay": float,
    "OpportunisticGraftTicks": int,
    "OpportunisticGraftPeers": int,
    "GraftFloodThreshold": float,
```

```
    "MaxIHaveLength": int,
    "MaxIHaveMessages": int,
    "MaxIDontWantLength": int,
    "MaxIDontWantMessages": int,
    "IWantFollowupTime": float,
    "IDontWantMessageThreshold": int,
    "IDontWantMessageTTL": int
  }
}
```

SubscribeToTopic

```
{
    "type": "subscribeToTopic",
    "topicID": str
}
```

SetTopicValidationDelay

```
{
  "type": "setTopicValidationDelay",
  "topicID": str,
  "delaySeconds": float
}
```

3.3 Major Changes

Although the initial framework proved to be a useful starting point, it was quickly apparent that it had been designed with very simple experiments in mind. As we increased the complexity of our simulations, we faced multiple limitations that had to be addressed along the way.

3.3.1 Time Granularity

The existing **WaitUntil** instruction did not provide sufficient granularity for fast-paced experiments. The choice of **elapsedSeconds** requires instruction timing to be aligned to full seconds. This did not let us create scenarios in which multiple messages are published by the network each second. In order to solve this, we replaced the field with **elapsedMillis**, which provides timing on a millisecond scale.

```
{
  "type": "waitUntil",
  "elapsedMillis": int
}
```

3.3.2 Node-Specific Instructions

Having a single **params.json** file shared across all nodes made scaling difficult in multiple ways. Most importantly, memory consumption grew quadratically with simulation size, as a larger node count would increase the total numbers of instructions, which would then need to be loaded into memory by every single process. Some scenarios required hundreds of thousands of individual instructions, resulting in gigabytes of memory usage per process just to store them. This severely limited the number of nodes we could simulate. It also made for a very slow startup as every node process had to parse the entire file.

There were three solutions we considered to solve this bottleneck. One strategy was to increase the complexity of the building blocks. With more expressive instructions that can define loops or other complex actions in a single JSON object, we would be able to cut down the number of instructions significantly. In the interest of facilitating support for additional messaging protocols in the future, however, we wanted to keep the complexity of the test harness minimal, so this approach fell out of favor. Another option was to rewrite the test harness such that it parses the JSON file as it processes the instructions. We decided against this for similar reasons. The logic would have to be re-implemented in every language a test harness might be provided in. Additionally, this had the potential to introduce other issues as the file would need to be kept open by all processes during the entirety of the simulation run. Finally, we decided to create a separate instruction file for each node process. The implementation of this is entirely contained in the setup script, which does not create additional work for future integrations.

The main downside of this approach is that instructions that apply to all nodes, such as most uses of **WaitUntil**, need to be duplicated across every single instruction file. However, large instruction file sizes could mostly be attributed to a high number of node-specific **Publish** instructions, which only need to be written once. In addition to nodes only having to parse their own instructions, this structure also allowed us to eliminate **IfNodeIdEquals** instructions, further reducing instruction complexity. The updated pipeline architecture can be seen in 3.2. The new structure has the additional advantage of making node changes in the middle of a simulation easy to implement. Once a node is supposed to shut down, its instruction file simply ends. With a shared instruction file, this would require either additional control flow primitives to exclude a specific set of

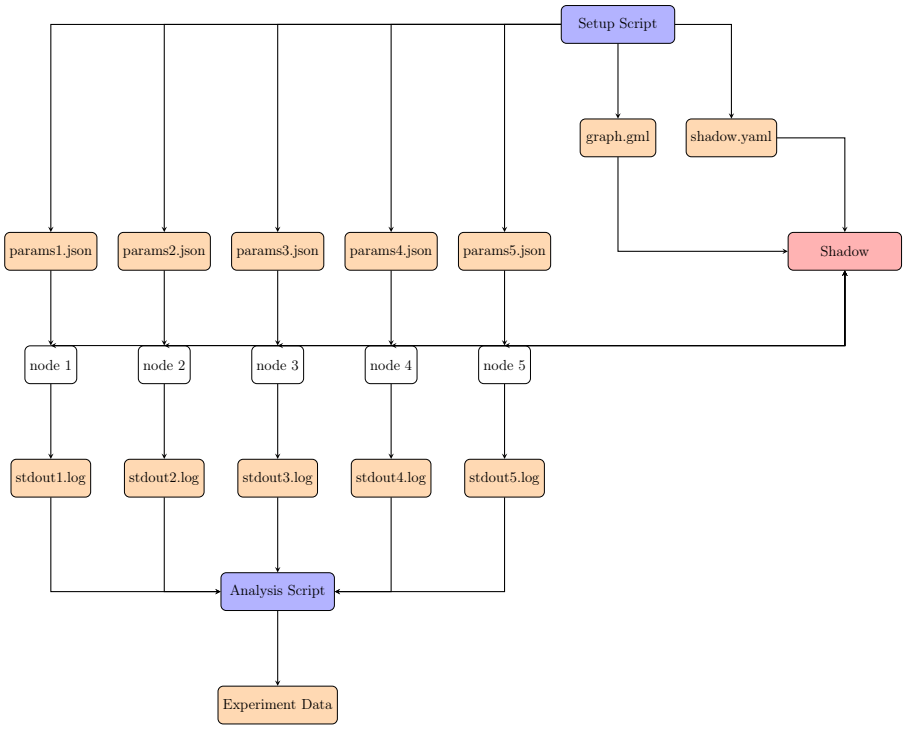


Figure 3.2: Updated Simulation Pipeline

nodes or an explicit **Shutdown** instruction.

The instruction set could be improved further by adding an explicit time field to instructions instead of specifying it indirectly through **WaitUntil**. This change would eliminate another instruction type and remove the majority of duplication across the node instruction files. We did not end up implementing this as the initial improvements were sufficient to successfully conduct our experiments.

3.3.3 Event Log Analysis

While the initial framework did have capabilities for rudimentary message delivery analysis, the collected data was not particularly useful for our purposes. We ended up fully replacing the data analysis module with a new Python script[16]. As the methodology for data collection is much less interesting than the actual data collected, its discussion in this chapter will be limited to the technical challenges involved. We faced an interesting scaling issue on this front too, as large simulations ended up producing close to a billion total lines of event logs across all node processes, much too large to fit into memory at once. The approach of parsing files one line at a time we had briefly considered for instructions in the

[16] https://github.com/lweitzendorf/gossip-shadow/blob/main/analyze_logs.py

test harness ended up being a good fit here. The analysis script keeps the next timestamp from each log file on a min heap, then advances the file on top of the heap to achieve a global chronological order while only having to keep one line per file in memory at a time.

3.3.4 Integrating Dynamic Optimal Graph

Beyond the previously listed general improvements to the framework, a significant amount of the time was spent adding functionality to run DOG with a compatible test harness. For the protocol itself, we ended up choosing **libp2p-dog**[17] as the base implementation. Besides being more self-contained than the original implementation that would have to be extracted from CometBFT, it also offers small improvements on some aspects of the base protocol [2] . It being implemented in Rust allowed us to reuse some parts of the existing test harness for **rust-libp2p Gossipsub**[18]. Most of the remaining integration work was focused on tracing all relevant events.

With both GossipSub and DOG integrated into a framework that can handle large amounts of simulation instructions and log files, we had all the necessary tools at our disposal to run the experiments.

[17] https://github.com/informalsystems/libp2p-dog

[18] https://github.com/libp2p/test-plans/tree/master/gossipsub-interop/rust-libp2p/src

Chapter 4

Evaluation

This chapter details all the experiments conducted within the scope of this thesis. For each experimental setup, we will go through specific parameters, configurations and logical stages of the simulation scenario. We will then discuss the collected data and how it does or does not answer the questions posed by our stated motivation.

The experimental setups will first establish a performance baseline for both protocols. We will then compare our results with previous work conducted by Bastien Faivre [2] before expanding on it through the simulation of adversarial conditions. The description of the first experiment introduces methodology that is relevant for all further sections of this chapter.

On account of DOG not being a pubsub protocol, we opted to instruct GossipSub nodes to subscribe to a single common topic in all scenarios to match simulation conditions as closely as possible.

All of our simulations were run on a machine with an **AMD EPYC 7313P** processor with **16 cores** and **256GB of RAM**. Shadow was configured to use 24 threads. Note that use of the machine during this time was not exclusive to our simulations. Given Shadow's deterministic discrete event model however, moderate CPU utilization by other tasks is not expected to affect the results in any way.

4.1 Baseline Simulations

4.1.1 Motivation

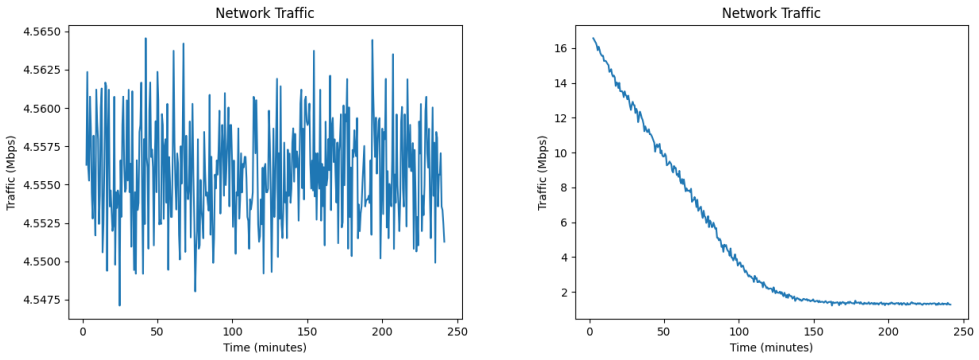
The aim of this initial experiment is to establish a baseline of protocol performance in non-adversarial conditions. We observe the bandwidth usage and message delivery latency in an unconstrained environment.

4.1.2 Setup

The simulation contains **1000 nodes**, which are each assigned **20 peers at random**. After an initial **120 second warm-up** to give the nodes time to establish connections, a **random node** is chosen **every 12 seconds** to publish a **1024 byte message**. There are **1200 messages** in total, implying a total **runtime of 4 hours** for the publish phase. The specific choice of 12 second intervals was intended to match the slot time of Ethereum mainnet[1], but other message intervals within a few multiples would have likely worked just as well. For Shadow, we used a network configuration with uniform **100 ms** latency and **1024 Mbps** symmetric bandwidth as well as the **real world configuration** described in 3.2.3. These configurations will be referred to as "Uniform" and "Real World", respectively.

4.1.3 Results

The first metric we look at is network traffic. More specifically, we consider the **bandwidth** that is **consumed** by the protocol **to send message payloads**. We measure the data rate in bits per second by taking **snapshots of cumulative traffic volume at 1 second intervals** throughout the **difference in data volume** by the d**ifference in time between two consecutive data points**. In this experiment, each payload will be exactly 1024 bytes as specified in 4.1.2. When looking at the observed network traffic throughout the entire experiment as shown in 4.1, we immediately see a stark difference in behavior between GossipSub (4.1a) and DOG (4.1b).



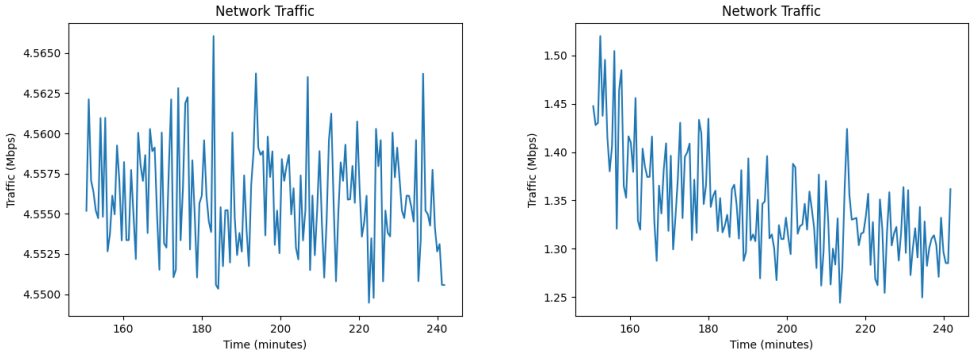
(a) GossipSub (b) DOG

Figure 4.1: Observed Network Traffic with Uniform Network

This difference can be explained by fundamental protocol design. Recall

[1] https://github.com/ethereum/consensus-specs/blob/master/specs/phase0/beacon-chain.md#time-parameters-1

chapter 2, which explains that DOG initially operates like floodsub and will forward messages to all of its peers. Given that it is assigned 20 of them, we would see an initial **message amplification of 20**. This slowly reduces over the course of the experiment and seems to reach a **steady state after 150 minutes**. With DOG's default redundancy target of 1.0, we would expect a bandwidth reduction of 90% as the delivery count of each message reduces from 20 to 2, which is indeed the case. GossipSub, on the other hand, immediately samples a smaller subset of the 20 peers it is connected to. With mesh target size defaulting to 6 as stated in 2.1, we would expect GossipSub to consistently use three times the bandwidth of DOG at its redundancy target. Also here, the collected data matches our observations. 4.2 provides a closer view that compares the bandwidth of both protocols in their steady state.



(a) GossipSub (b) DOG

Figure 4.2: Observed Network Traffic with Uniform Network in Steady State

Remark 4.1. Let us quickly verify that the absolute bandwidth values observed are in line with what we would expect given the simulation scenario. Recall that there is a 1024 byte message sent to 1000 nodes every 12 seconds. This means that with a perfect message amplification factor of 1, we expect to see an average network traffic of

$$\frac{(1024 \text{ bytes}) \cdot (1000 - 1)}{12\text{s}} = 85,248 \frac{\text{bytes}}{\text{s}} = 681,984 \frac{\text{bits}}{\text{s}} = 0.681984 \text{ Mbps} \tag{4.1}$$

This is indeed closely in line with the observation, given GossipSub and DOG's respective expected amplification factors of 6 and 2.

Figure 4.3 provides another view of this, directly comparing the observed network traffic against the calculated optimal bandwidth.

Let us now briefly consider the question of whether bandwidth usage is uniform across nodes. At first glance, the answer seems to lean towards the affirmative for protocols, with a multiple of 3 between the nodes with the highest

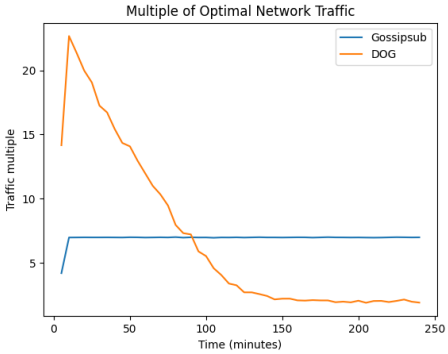
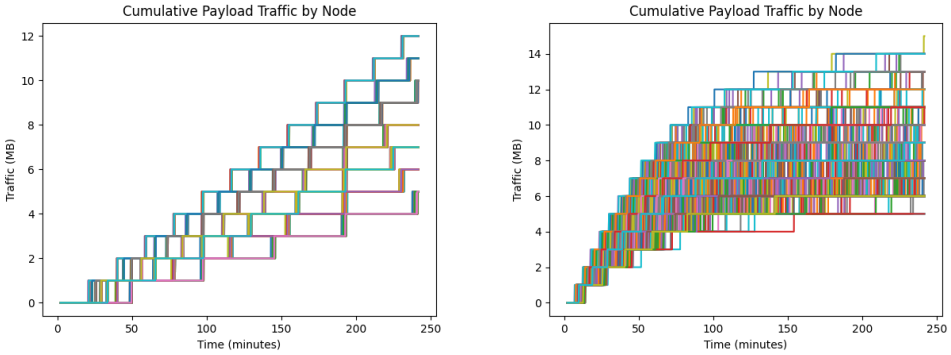


Figure 4.3: Observed Traffic Multiple over Optimum with Uniform Network

and lowest bandwidth usage across the entire simulation, as can be seen in 4.4. However, considering steady state in isolation paints a different picture. DOG initially exhibits perfect uniformity and diverges further as it establishes a more efficient routing table. In its steady state, bandwidth usage of DOG is skewed more strongly towards a small subset of the network than in GossipSub. We consider this result somewhat surprising in an environment with uniform latency and random peering across many nodes.

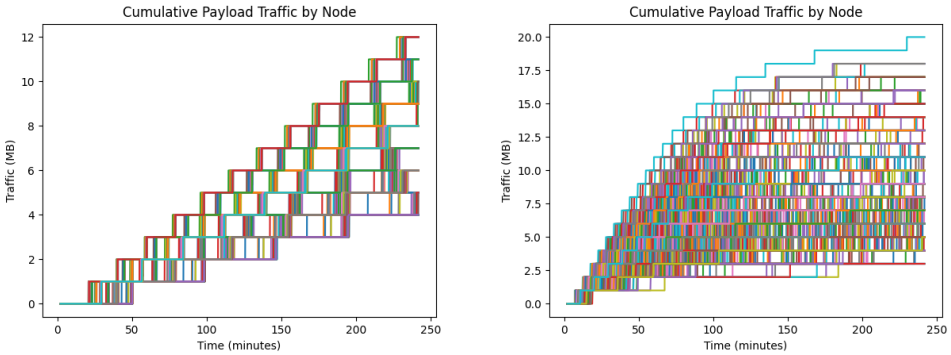


(a) GossipSub (b) DOG

Figure 4.4: Observed Total Traffic by Node with Uniform Network

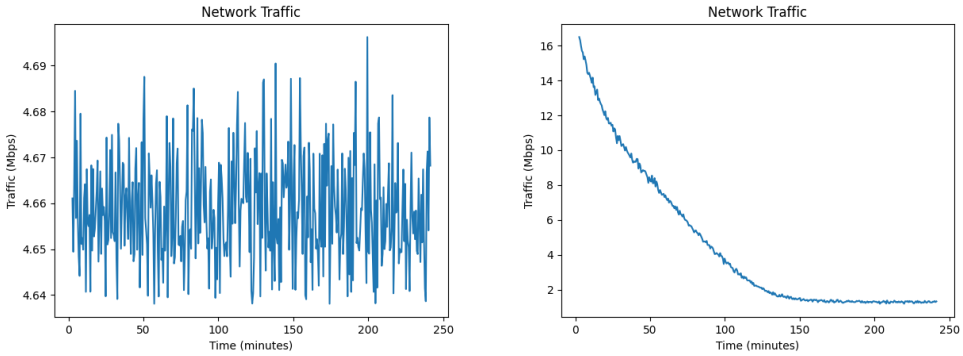
To investigate this further, we look at data for the same simulation scenario with the real world network configuration, which possesses a wider range of latencies. Figure 4.5b shows the same general behavior, which is to say that initial uniformity of DOG decays throughout the simulation. GossipSub exhibits a constant spread, covering an approximate multiple of three. However, we can see that DOG diverges even further with this network configuration as it presumably converges on a small set of low latency routes. Note that the overall network traffic of the real world configuration is practically identical to that of the uniform

latency configuration. This is shown in 4.6.



(a) GossipSub (b) DOG

Figure 4.5: Observed Total Traffic by Node with Real World Network



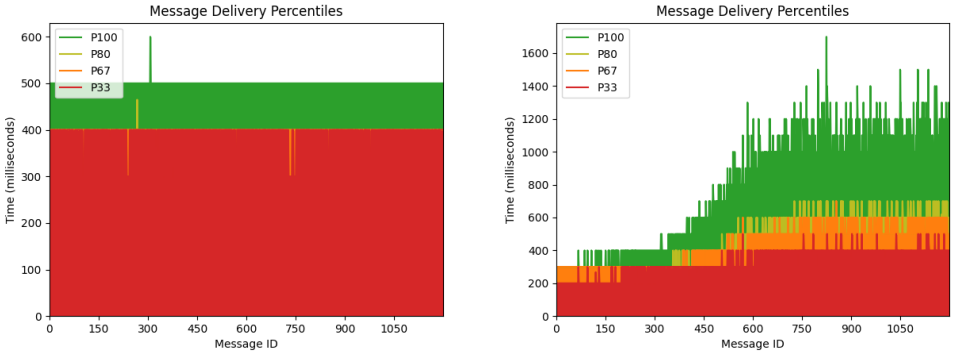
(a) GossipSub (b) DOG

Figure 4.6: Observed Network Traffic with Real World Network

Having established clear differences in bandwidth consumption between DOG and GossipSub, we move on to the dissection of message latencies. For simple experiments such as this, the set of nodes remains constant throughout the simulation, allowing us to take measurements over the list of all nodes. We first observe delivery data for the uniform latency environment, illustrated in 4.7. We plot the 33rd, 67th, 80th, and 100th percentile for each message, marking the amount of elapsed time needed from initial message publication for the respective percentage of the network to have received it. Given the uniform communication latency and lack of processing or bandwidth constraints in this scenario, the data also allows us to cleanly determine the number of network hops by dividing by the 100 ms latency.

In 4.7, we observe another significant difference in behavior between the two

protocols. GossipSub maintains a consistent latency throughout the simulation, reaching 80% of the network after 400 milliseconds, or 4 hops, most of the time and almost always reaching the remaining 20% within a total of 5 hops. Again, DOG's behavior changes as it adapts its routing table. When connected to all 20 peers, it achieves full network permeation within only three hops. This increases significantly as the number of routes is reduced. While DOG manages to reach two thirds of the network within 5 to 6 hops, it exhibits a rather long tail. Up to 17 hops are needed to reach the entire network, with 10 being required for the majority of messages. This maybe be explained by the lower potential for optimized routing to compensate for a low number of active peers in an environment with uniform latency.



(a) GossipSub (b) DOG

Figure 4.7: Observed Message Delivery Percentile Latency with Uniform Network

Remark 4.2. Again, we should briefly verify that the observed values are plausible given the simulation conditions. Consider the network as a simple graph with 1000 nodes and edges representing connections between peers. In GossipSub's case, we have an average out-degree of 6. Under ideal conditions, this means we can reach all nodes with 4 hops. If we simply model the accompanying amplification factor by multiplying the total number of nodes to be reached, we arrive at the observed 5 hops.

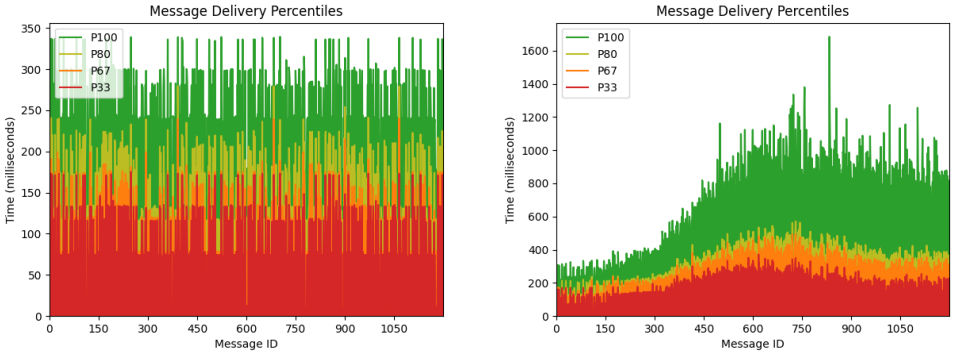
$$\lceil \log_6(1000) \rceil = 4, \quad \lceil \log_6(6 \cdot 1000) \rceil = 5 \tag{4.2}$$

The same model predicts 11 hops for Dynamic Optimal Graph with its eventual out-degree of 2, which is roughly in line with the observed average for 100th percentile delivery in steady state.

$$\lceil \log_2(1000) \rceil = 10, \quad \lceil \log_2(2 \cdot 1000) \rceil = 11 \tag{4.3}$$

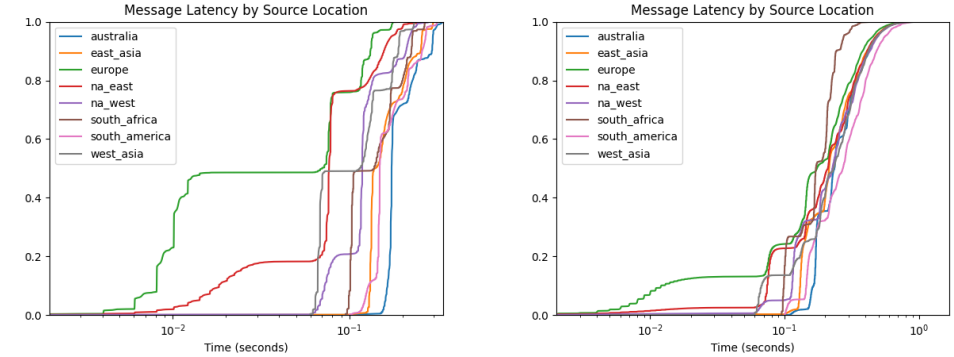
We now evaluate latency data in the real world configuration to investigate whether this allows DOG to play more to its strengths. In general, we see the

same trends here as we do with the uniform network configuration. 4.8b shows that DOG achieves full network saturation within 250 to 350 ms, which slowly deteriorates to around a second of average P100 latency. Surprisingly to us, GossipSub's advantage seems to increase in this configuration. 4.8a shows that the protocol consistently manages to reach the entire network within 350 ms. Furthermore, P67 latency is below 200 ms for almost all messages. 4.9 provides further context for how individuals subsets of the network are saturated.



(a) GossipSub (b) DOG

Figure 4.8: Observed Message Delivery Percentile Latency with Real World Network



(a) GossipSub (b) DOG

Figure 4.9: Observed Message Delivery Density by Location in Real World Network

We speculate that DOG's indirect way of preserving low latency connections breaks down in cases where the source of published messages frequently changes or, more generally, peer behavior is inconsistent. GossipSub includes a mechanism to model long-term behavior of other nodes via its peer scoring (chapter 2), whereas DOG does not. Once the Dynamic Optimal Graph router observes a

late message and decides to disable its corresponding route, there is no way for it to be re-enabled on merit. This means that slight inconsistencies in performance have the potential to lead DOG towards a heavily compromised routing table. While we did not investigate this further, it is our most plausible explanation for the observations at hand.

4.1.4 Discussion

The experiment shows that with both protocols in steady state, configured to their respective defaults, there is a clear trade-off between latency and bandwidth consumption. If given sufficient time to stabilize, DOG uses roughly twice the theoretical minimum bandwidth to disseminate messages, as its redundancy target of 1.0 would suggest. GossipSub consumes three times as much, i.e. 6x the theoretical minimum, but in doing so manages to deliver messages significantly faster.

The latter result is particularly interesting because it contradicts the findings of Faivre [2], whose implementation of Dynamic Optimal Graph we used for our experiments. In their work, they compare GossipSub and DOG on 32 real machines and find that DOG exhibits significantly lower latency than GossipSub. Further, they propose improvements to routing in DOG that would further entrench existing issues if our posited reasoning for the observed suboptimal routing were to be true.

These results pose three new questions, which we seek to answer in follow-up experiments:

1. Do our results resemble those of Faivre more closely if we replicate additional details of their experimental setup in our simulation? (Section 4.2)
2. How do the performance characteristics of these protocols change if we configure them to run at the same message amplification factor? (Section 4.4)
3. Can DOG's requirement for extensive stabilization time be used against it in a low bandwidth environment? (Sections 4.3, 4.5)

4.2 Disagreement With Previous Work

4.2.1 Motivation

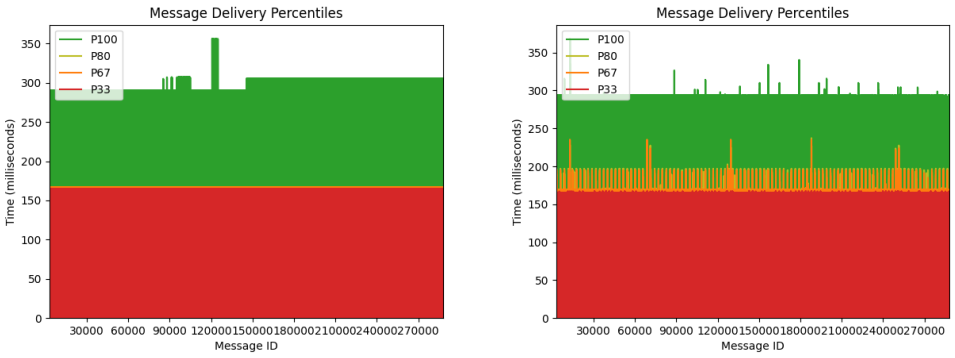
Given that the results of our baseline simulation diverge from the work done by Faivre [2], we aim to provide a closer match to their experimental setup with this scenario and reevaluate whether our findings agree.

4.2.2 Setup

The scenario consists of **32 nodes**, which are each assigned **10 random peers**. Every node is instructed to send **30 1024 byte messages per second**, resulting in **960 messages per second across the entire** network. As this simulation is resource-intensive to run, the scenario only lasts **5 minutes**. In this time, a total of **288,000 messages** is published. We use the **real world network configuration**, with **10 GBps of symmetric bandwidth** to match the servers used by Faivre. [2]

4.2.3 Results

We observe no significant differences in message latency between the two protocols in this scenario. Figure 4.10 shows particularly consistent delivery times, potentially attributable to the low number of nodes. To confirm this, we look at message latency by location in figure 4.11, which seems to substantiate this initial impression.



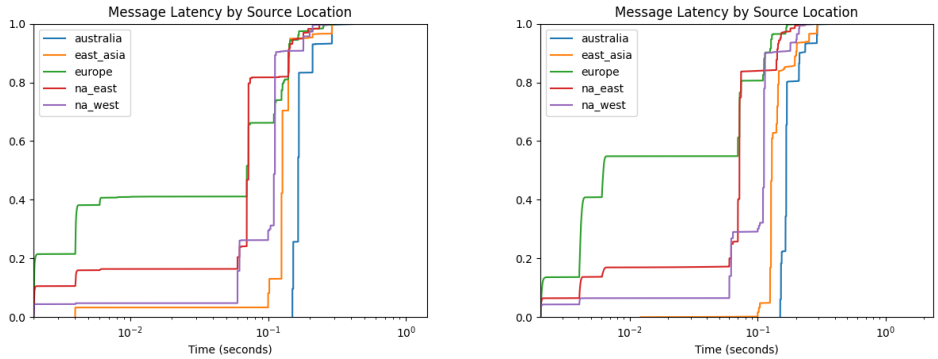
(a) GossipSub (b) DOG

Figure 4.10: Observed Message Latency Percentiles

Faivre claims a global workload of just under 1 MB per second [2]. However, this is not very useful as it only considers the bandwidth required to publish, not to disseminate the message. Considering the same assumptions for theoretical minimum bandwidth we did before, we arrive at approximately 244 Mbps.

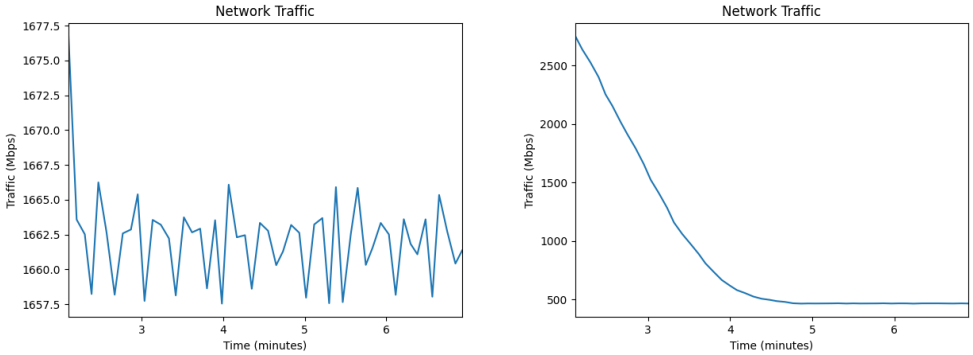
$$\frac{30 * 32 * 31 * 1024 \text{ bytes}}{\text{s}} = 243.79392 \text{ Mbps} \tag{4.4}$$

In relation to this, we can also observe similar amplification factors to previous scenarios in figure 4.12.



(a) GossipSub (b) DOG

Figure 4.11: Observed Message Latency by Location



(a) GossipSub (b) DOG

Figure 4.12: Observed Network Traffic

4.2.4 Discussion

Even after specifically trying to match the experimental setup of Faivre, we still do not see similar results. In their work, Faivre cites GossipSub's lazy pull behavior to explain the lower performance. We think this is unlikely as a network of only 32 nodes should be able to be saturated within only two hops, given a default mesh size of 6. Although we cannot give a definitive answer as to which results should be considered more valid, we believe that a large-scale simulation is more representative than 32 servers with virtualized latency via the Linux traffic control tool [2].

4.3 Attacking DOG

4.3.1 Motivation

As mentioned in both chapter 2 and section 4.1, Dynamic Optimal Graph behaves like a naive floodsub implementation on initialization and takes significant time to converge on its steady state. Further, DOG does not have any inherent limits to the number of connections it will accept. As such, it is reasonable to ask whether it is also vulnerable to some of the same attacks as floodsub.

In this scenario, we place an initial set honest nodes in a bandwidth-constrained environment. After they have reached steady state, a small set of malicious nodes joins the network, attempting to connect to everyone else in an attempt to increase bandwidth beyond their limits, causing latency to spike.

4.3.2 Setup

This is the first scenario in which the set of participating nodes changes throughout the simulation. In order to manages this, we split the **experiment** into explicit **phases**. Each phase has static set of nodes. It begins with a set amount of **warmup time** in order to let new nodes joining the network start up. This is followed by the **publish sub phase**, in which messages are sent. Finally, there is some **cooldown time** to prevent messages that are still being disseminated from leaking into the next phase. This clearly defines which nodes participate in which part of the scenario, which is important to have well-defined latency metrics.

We start the scenario with an active set of **50 nodes**, each of which is assigned **8 random peers**. After an initial 120 second startup phase. This is followed by a set of **30 phases**, each of which consists of a 60 second warmup, **250 messages** sent in **100 ms intervals** by **randomly selected nodes**, each **4400 bytes** in size, followed by a 60 second cooldown. This is enough time for both protocols to reach their steady state.

Next, we add **8 new nodes**, each of which is instructed to **connect to everyone else** in the network, forming up to $\mathbf{8 \times 57}$ **connections**. This is then followed by another **15 phases** with 60 seconds of warmup, 250 messages and a longer 5 minute cooldown to account for some of the messaged delayed by the induced bandwidth constraints.

In terms of network configuration, each node is restricted to **20 Mbps** of bandwidth for both upload and download. Communication latency is uniformly set to **20 ms**.

In order to collect data for this experiment, we had to carefully balance the attack such that it induces enough of a bandwidth constraint to produce latency

spikes, but not to the extent that the buffer of undelivered network packets within Shadow would exceed our simulation host's system memory. This is strictly an inherent limitation with such a holistic simulation environment, which has to not only virtualize all node processes, but also the routing infrastructure between them, on a single machine.

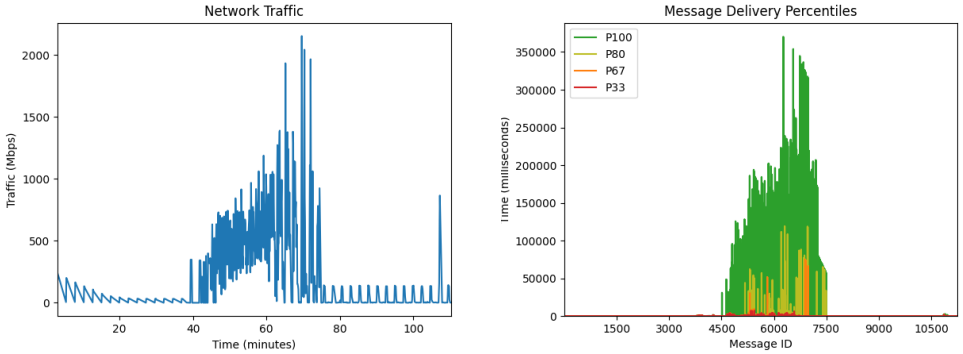
It would likely be possible to simplify the experimental setup by combining phases. However, as it took a lot of iteration to arrive at the current state and simplifying the scenario would not have a major impact on its result, this was not a priority to address.

4.3.3 Results

Due to the phase-based structure of this experiment, time-based graphs are less continuous because of the warm-up and cool-down sub phases. An example of this can be seen in figure 4.13a. Each bandwidth spike represents a separate phase. Looking at the chart, one can still clearly discern the DOG initialization, in which it settles into a steady state as in experiments prior.

As the new nodes enter the network, bandwidth usage spikes significantly. This in turn causes message latency to spike, increasing P33 latency to as high as 10 seconds while P67 latency exceeds a full minute for a handful of messages.

An important detail is that message latencies are only collected over honest nodes. This excludes messages sent by honest nodes and delivered to malicious nodes and messages sent by malicious nodes. By only considering honest nodes, we ensure that an attempted attack is not mistakenly considered successful in the cases where malicious nodes only manage to degrade their own performance.



(a) Network Traffic (b) Message Latency

Figure 4.13: Observed Payload Traffic and Message Latency for DOG

However, this attack is particularly bandwidth-intensive for the attacker. Figure 4.14 shows a comparison of cumulative traffic for each node in this scenario.

The 8 lines on top with the strongest traffic spike represent the 8 malicious nodes. Although these nodes need to be able to handle a lot more bandwidth than anyone else in the network, they importantly manage to also significantly increase bandwidth consumption for everyone else.

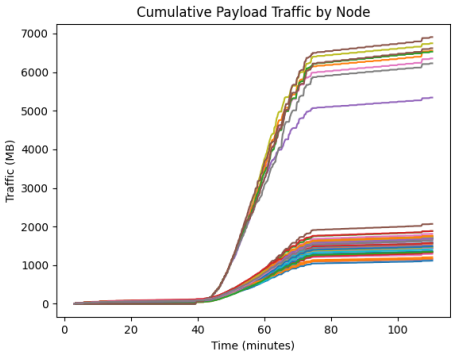
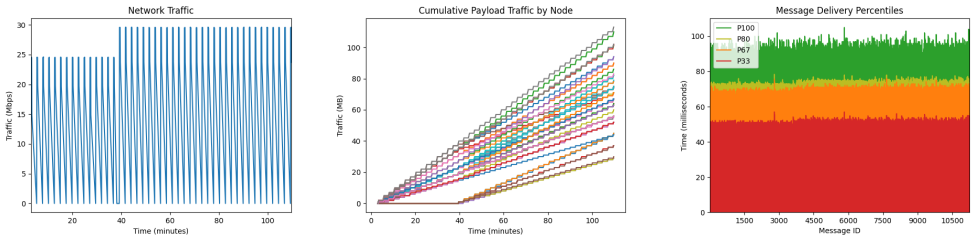


Figure 4.14: Observed Network Traffic by Node for DOG

Figure 4.15 provides a stark contrast by showcasing the same set of metrics for GossipSub. Due to its mesh construction, the large number of connection requests does not have a significant impact. In 4.15a, one can discern what amounts to an approximate 20% increase in total network traffic at the time of attack. However, this is simply due to the increase in number of participating nodes from 50 to 58. 4.15b further substantiates this, as the rate of change in the cumulative payload does not change significantly at any point in the simulation. Similarly, message latencies do not display any discernible attack impact as can be seen in figure 4.15c.



(a) Network Traffic (b) Traffic by Node (c) Message Latency

Figure 4.15: Observed Metrics for GossipSub

4.3.4 Discussion

In summary, we have managed to demonstrate a denial of service attack vector in DOG by adding a small (16% of honest nodes) number of malicious nodes

to the network. The attack itself simply consists of sending a large number of connection requests to honest network participants. The result of the attack is a major disruption to message dissemination, increasing 67th percentile latency far beyond what would usually be considered acceptable in a production deployment. Although the attack in its current form is limited in time, it can be trivially extended by disconnecting and re-connecting the malicious nodes.

A mitigation of this issue is possible, but potentially not trivial. One might think of simply limiting the number of active connections to avoid this. However, a naive implementation of such a restriction might lead to new issues. DOG relies on initially having a large number of routes at its disposal, so it can construct an effective routing table from them. If the number of active connections is restricted too tightly, it is likely to affect DOG's routing performance. A low limit to the number of peers can also make a node vulnerable to eclipse attacks [6].

A more reasonable approach might be to impose a rate limit on new connections, optionally subject to a delay after startup to not slow down the process of joining the network. This would make it much more difficult for an attacker to overwhelm other nodes and has less potential for unintended side effects.

4.4 Matching Amplification Factor

4.4.1 Motivation

The aim of this experiment is to investigate whether the tradeoff between latency and bandwidth we observed in our baseline experiments is inherent to protocol design to some degree. We are interested in whether GossipSub, if set to match DOG's default message amplification factor of 2, still remains functional, and if so, how it affects latency. Conversely, we want to see whether DOG, if set to match GossipSub's default amplification factor of 6, is able to effectively make use of the additional bandwidth budget to improve message latency.

4.4.2 Setup

The simulation scenario is similar to that of the baseline simulation in section 4.1. There are **1000 nodes** with **20 randomly assigned peers** each. The scenario starts with a **120 second warm-up** followed by **1200 1024 byte messages published every 12 seconds by a random node**. The network configuration was set to a **uniform 100 ms latency** and **1024 Mbps symmetric bandwidth**.

We configure **GossipSub** with the following **parameters**:

$$D = 2,\ D_{low} = 1,\ D_{high} = 3,\ D_{score} = 2,\ D_{out} = 0 \tag{4.5}$$

For DOG, we simply set the **target redundancy** to **5.0**.

4.4.3 Results

First, let us examine the performance of DOG with its increased target redundancy. Looking at figure 4.16, we can confirm that its steady state network traffic indeed closely matches that of GossipSub in its default configuration (4.1a) now.

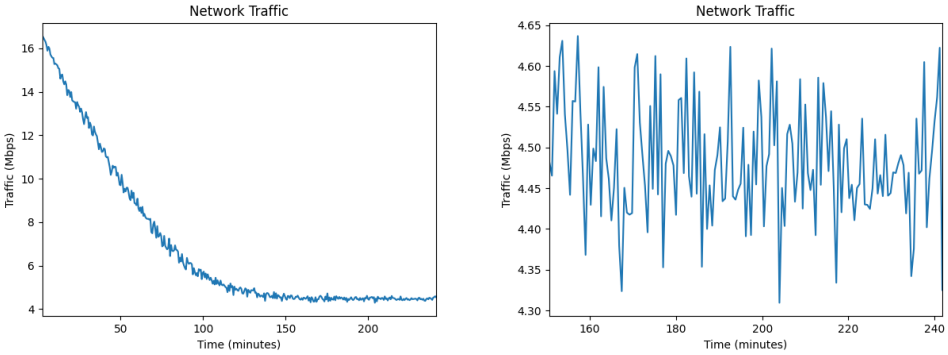
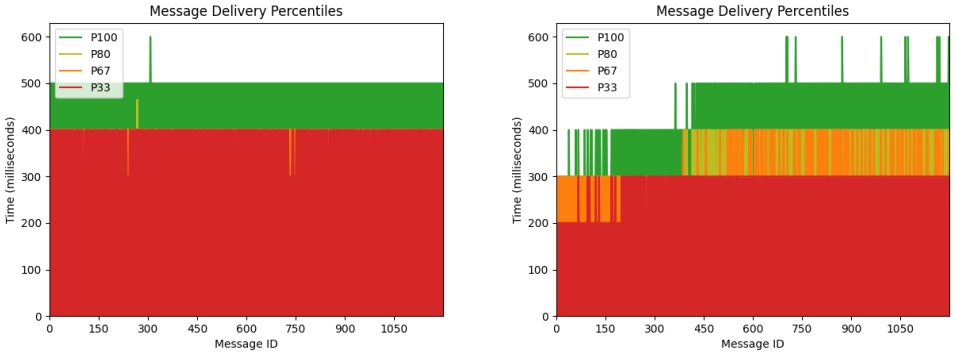


Figure 4.16: Observed Network Traffic for DOG with Amplification Factor 6

Further, we observe very similar latency performance when looking at figure 4.17, which contrasts the earlier baseline GossipSub result with those of DOG in its high amplification configuration. Due to the nature of the protocol, DOG initially exhibits higher bandwidth consumption and better latency again, before settling into a state that is very similar to that of GossipSub under these conditions.



(a) GossipSub (b) DOG

Figure 4.17: Observed Message Latency with Amplification Factor 6

The results of the converse adjustment are equally relevant. Figure 4.18

shows GossipSub now matching the bandwidth we saw from DOG in its default configuration. Unfortunately, the same cannot be said about performance. As shown in 4.19, GossipSub exhibits even higher latency than DOG, requiring 2.5 seconds to reach the entire network of 1000 nodes.

Intuitively, this seems to make sense. In an unconstrained simulation environment such as in this scenario, it is likely that all nodes perform well enough to stay in a peer's mesh once grafted, which effectively reduces mesh selection to pure random sampling. This, coupled with the fact that we are severely restricting the size of the mesh with a target of 2 and tight bounds means that the expected peer graph resulting from this process is not particularly suitable for fast routing.

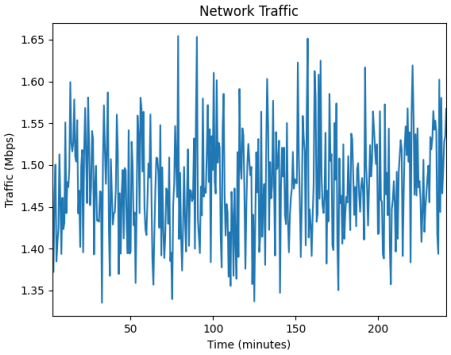
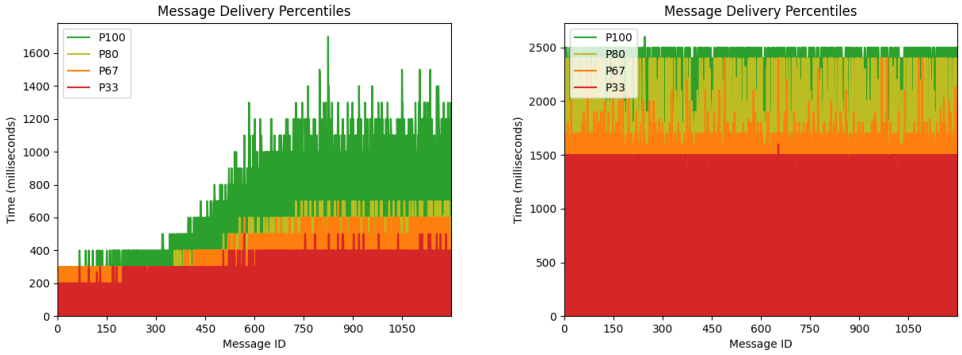


Figure 4.18: Observed Network Traffic for GossipSub with Amplification Factor 2



(a) DOG (b) GossipSub

Figure 4.19: Observed Message Latency with Amplification Factor 2

4.4.4 Discussion

Judging by the results of our simple experiment, DOG is the more versatile protocol with regard to configuration options. In a stable environment where average traffic volume is a constraint, DOG will be the better choice as GossipSub struggles to maintain efficient routing when placed under such heavy constraints. However, a solution for this might be as simple as weighing message latency more heavily. The results of experiment 4.6 suggest that GossipSub is able to reconfigure itself to optimize latency in some situations.

In the other direction, DOG's bandwidth consumption can be granularly adjusted up from its default amplification factor to meet more stringent performance expectations. Note that this result does not yet consider any adversarial network participants as part of the comparison.

4.5 Adaptation to Node Churn

4.5.1 Motivation

After having seen Dynamic Optimal Graph's vulnerability to large numbers of connection requests, this experiment poses the question whether general churn, i.e. the process of nodes leaving and joining the network might also impact the protocol's ability to function effectively.

4.5.2 Setup

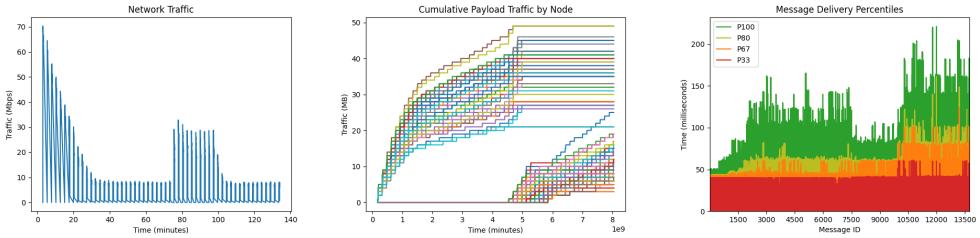
We start with an active set of **50 nodes**, each randomly assigned **10 peers**. Identically to the previous attack scenario, the experiment commences with an initial **120 second startup phase** before moving on to **30 phases of 250 1024 byte messages each** to let the protocols reach their steady state. Each of these phases is bookmarked by a 60 second warm-up and 60 second cool-down.

We subsequently impose a rather heavy churn through 10 additional 250 message phases, at the end of which we shut down 10 of the active-at-the-time nodes and replace them with 10 new ones. Each of these new joiners is also assigned 10 random peers from the active set of the subsequent phase. This churn rate implies a total of 150 distinct nodes throughout the simulation. We end the experiment with **15 standard phases** with 250 messages each.

We reuse the previous network configuration, which imposes a symmetric **20 Mbps** bandwidth limit and **20 millisecond latency**.

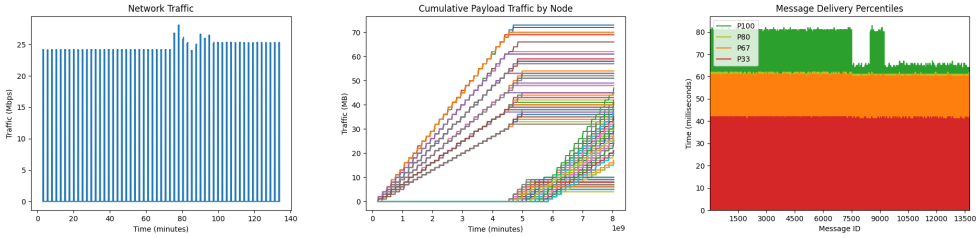
4.5.3 Results

For this scenario, we will primarily focus attention on the changes in total network traffic. Figure 4.20a showcases a clear pattern. Following the by-now familiar transition to steady state, there is a distinct spike in bandwidth usage as the churn commences. Each new phase produces an initial spike from which the network slowly recovers before moving on to the next phase. For message latency data, observe figure 4.20c. We can see that after the initial steady state, latency slightly improves as the churn temporarily increases the number of active connections. After the churn subsides, DOG seems to converge on a less efficient routing table compared to its pre-churn state and message latency sees a moderate increase. For comparison, equivalent data for GossipSub is shown in figure 4.21. For the most part, we observe similar behavior as in previous scenarios. GossipSub remains mostly stable, with no significant changes in global metrics as the churn phases begin and end. Interestingly, GossipSub manages to decrease its P100 latency by one hop after the churn phases.



(a) Total Network Traffic (b) Traffic by Node (c) Message Latency

Figure 4.20: Observed Metrics for DOG



(a) Total Network Traffic (b) Traffic by Node (c) Message Latency

Figure 4.21: Observed Metrics for GossipSub

4.5.4 Discussion

Regular churn seems not to pose an issue for either protocol. While bandwidth usage does increase, it never goes beyond the rate of traffic flow required in its

initial state. One could make an argument that even approaching these levels is problematic given that an environment in which DOG is deployed for its bandwidth-saving properties would most likely rely on only a small subset of nodes starting up at any given time, whereas a bandwidth increase from churn, affects a much larger part of the network. On the other hand, the simulated churn is rather heavy-handed with 20% of the active node set leaving every 250 messages. It is also true that this churn actively decreases dissemination latency, which is why we do not see an issue with the protocol in this area.

4.6 Two Cliques

4.6.1 Motivation

In this scenario, we want to investigate how GossipSub behaves if its initial random sampling of full-message peers happens to place it within a degenerate topology with high-latency paths. The main question posed is whether GossipSub is able to recover and establish a more efficient message route on its own and if so, how quickly.

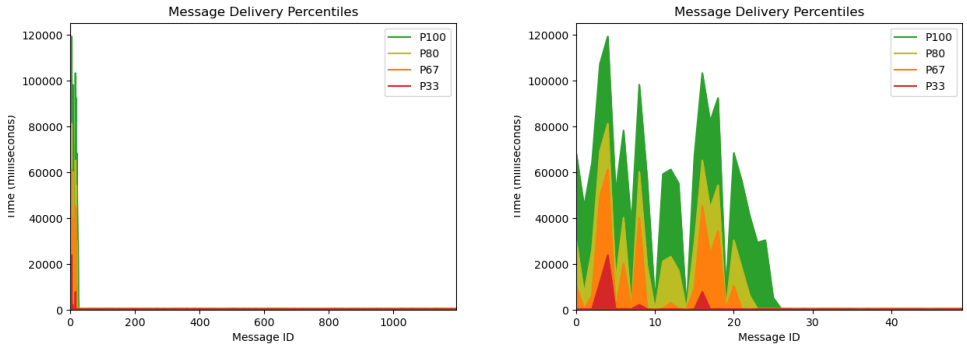
4.6.2 Setup

The experiment contains **1000 nodes**. Instead of assigning peers randomly, we construct a graph. We start by generating a **random graph G with 1000 nodes**, each with an an **expected degree of 6**. By virtue of this property, we ensure that the graph is suitable for efficient routing. Consider **two partitions of** G of equal size, G' and G''. We **add additional edges** to each partition to obtain a final graph H with **two 500-cliques and a 6-regular expander** as subgraph. Due to the large amount of added edges, GossipSub's mesh sampling is overwhelmingly unlikely to pick edges within the 6-expander. Each node is then instructed to connect to its respective neighbors in H.

The scenario is otherwise identical to the uniform baseline simulation in section 4.1. We send **1200 messages** in **12 second intervals**, each 1024 bytes in size.

4.6.3 Results

At the beginning of the simulation, we see GossipSub struggling to route within the dense graph as expected. Figure 4.22a shows the initial spike with message delivery taking up to two minutes. 4.22b provides a closer view for the first 50 messages in the simulation. By message 27, GossipSub seems to have already managed to graft enough new peers into its mesh to allow for efficient routing again.



(a) Full Simulation (b) First 50 Messages

Figure 4.22: Observed Message Latency Percentiles

Figure 4.23 tells a similar story. We see a strong spike in bandwidth consumption before GossipSub settles into its usual steady state at 6x to 7x the optimal bandwidth.

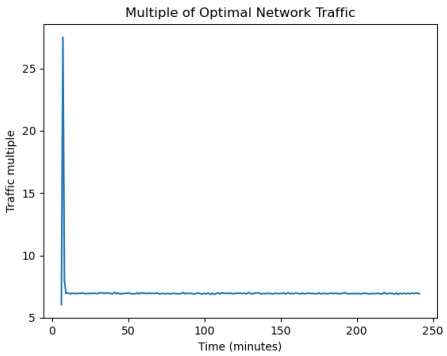


Figure 4.23: Observed Multiple of Theoretical Minimum Bandwidth

4.6.4 Discussion

GossipSub managed to very quickly reconfigure itself to reestablish sufficiently high routing efficiency. However, this poses new questions for us. How exactly did it decide to forego its initial peers and how did it find new ones so quickly? Can we do anything to prevent GossipSub from doing so or at least make it harder? These are interesting questions, but they require a deep dive into GossipSub-specific metrics such as peer scores and detailed analysis of the hundreds of millions of lines of logs this scenario produces. We did not find the time to answer these questions within the scope of this thesis, so they shall remain questions for another day.

Chapter 5

Conclusion

In this thesis, we have significantly expanded on an existing libp2p framework and demonstrated its functionality by conducting an extensive evaluation of two gossip-based message dissemination protocols, GossipSub and Dynamic Optimal Graph. In doing so, we were able to establish deep knowledge of both protocols and leverage this knowledge to successfully demonstrate a denial of service attack against a simulated Dynamic Optimal Graph network. Beyond these results, there are many aspects of our work than can be built and improved upon.

5.1 Future Work

As mentioned in subsection 4.3.4, the successful attack is interesting by itself, but it can be enhanced in a few ways. First, the current attack scenario can be upheld by disconnecting malicious nodes, once honest nodes have managed to significantly reduce their active connections, and reconnect them. Although this is simple to implement, it will require a meaningful number of iterations to determine a suitable interval in which to repeat the connection requests. Second, the attack is still somewhat costly to the attackers as they require a lot more bandwidth than the rest of the network. It is likely that at least the required upload bandwidth can be significantly reduced with a modified protocol client that does not forward all messages sent to the malicious nodes.

If run on a larger network, a smaller number of malicious nodes can likely achieve the same result. Note that doing this also increases the bandwidth requirement for the attacker even further.

It is also worth exploring whether DOG is vulnerable to other simple attacks as the protocol does not have many built-in protections. For example, its approach to replacing low-performance peers is based on message timing. Consistently delaying messages is likely to get an attacker dropped. However, if the attacker were to e.g. fully withhold messages, there is no opportunity for this condition to trigger.

In the simulation framework itself, there is also opportunity for improvements.

As an example, there is currently no straightforward way to deterministically assign different roles within a simulation. This could be useful to give an attacker more bandwidth or a separate binary to run than the honest nodes in the network. As already mentioned in chapter 3, the instruction set could also be further simplified.

Lastly, there are some loose ends with event tracing that need to be tied up. In the GossipSub implementation, the tracer does not include the size of a message in its event information. As this is part of the libp2p library code and most simulations used a message size of 1024 bytes, we simply made do with hardcoded message sizes so far. While the DOG test harness does have access to the message size, it only traces payload messages. Control messages such as **HaveTx** are not currently logged. Adding this would allow the analysis script to provide more fine-grained information about bandwidth usage.

Bibliography

[1] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks," *arXiv preprint arXiv:2007.02754*, 2020.

[2] B. Faivre, "Design and analysis of networking building blocks in malachite, a modern bft consensus engine," Master's thesis, École Polytechnique Fédérale de Lausanne, 2025.

[3] M. Jelasity, "Gossip," in *Self-organising software: From natural to artificial adaptation*. Springer, 2011, pp. 139–162.

[4] D. P. Bauer, "Filecoin," in *Getting Started with Ethereum: A Step-by-Step Guide to Becoming a Blockchain Developer*. Springer, 2022, pp. 97–101.

[5] J. Dinger and O. P. Waldhorst, "Decentralized bootstrapping of p2p systems: A practical view," in *International Conference on Research in Networking*. Springer, 2009, pp. 703–715.

[6] A. Singh, T.-W. Ngan, P. Druschel, D. S. Wallach *et al.*, "Eclipse attacks on overlay networks: Threats and defenses," 2006.