Refinement Based Reasoning of P2P Protocols is Feasible and Effective

by

Ankit Kumar

Submitted to the Khoury College of Computer Sciences
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

at the

NORTHEASTERN UNIVERSITY

April 2026

© 2026 Ankit Kumar. All rights reserved.

Authored by: Ankit Kumar

Northeastern University

April 2026

Certified by: Panagiotis Manolios

Professor, Thesis Supervisor

Accepted by: Amal Ahmed

Professor

Khoury College of Computer Sciences

Docusign Envelope ID: 76EBB6F1-0F22-8A13-8273-7FA51CF0B797

Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Thesis Title: **Refinement Based Reasoning of P2P Protocols is Feasible and Effective**

Author: Ankit Kumar

PhD Program: ✓ Computer Science ____ Cybersecurity ____ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

Signed by: Pete Manolios 31C112284C33478...	3/31/2026
Thesis Advisor	*Date*
Signed by: A9A010DAFCA2426...	3/31/2026
Thesis Reader	*Date*
Signed by: Cristina Nita-Rotaru E059A7AA25BE47B...	3/29/2026
Thesis Reader	*Date*
	4/3/2026
Thesis Reader	*Date*
Thesis Reader	*Date*

KHOURY COLLEGE APPROVAL:

	4/7/2026
Associate Dean for Graduate Programs	*Date*

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Pete Morency	4/8/2026
Recipient's Signature	*Date*

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

To Dadaji, Dadima, Nanaji, and Nanima

Ever grateful for your blessings

Refinement Based Reasoning of P2P Protocols is Feasible and Effective

by

Ankit Kumar

Submitted to the Khoury College of Computer Sciences
on April 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

ABSTRACT

Gossipsub is the message dissemination protocol underlying Ethereum, IPFS, and Filecoin—systems with a combined market capitalization exceeding $280 billion. Despite its critical role in Web3 infrastructure, Gossipsub lacks a formal specification; its security mechanisms have been evaluated primarily through ad hoc testing; and its operational parameters were selected heuristically, with limited quantitative justification.

This dissertation develops a compositional, refinement-based methodology for the rigorous analysis of peer-to-peer (P2P) publish-subscribe protocols and applies it to Gossipsub, making the following three principal contributions: (1) the first formal, executable specification of Gossipsub, now recognized as the official reference model; (2) the discovery and responsible disclosure of novel attacks against Gossipsub v1.1 based consensus layer of Ethereum (CVE-2022-47547); and (3) a systematic analysis demonstrating that Ethereum's deployed parameter configuration for Gossipsub is Pareto-dominated by leaner alternatives under realistic network churn.

We construct a refinement hierarchy spanning four protocol layers, each one building on top of the previous: Broadcastsub (an abstract specification of instant message delivery to all subscribing peers), Floodsub (a flooding-based realization over an underlying graph), Meshsub (controlled flooding via bounded-degree mesh overlays, and gossip to non-mesh neighbors), and Gossipsub v1.1 (scoring-based mesh management to guard against misbehaving peers).

We use a synergistic combination of theorem proving, property-based testing and counterexample generation to analyze this hierarchy. We prove that Meshsub implements Floodsub and Floodsub implements Broadcastsub; by compositional refinement, Meshsub implements Broadcastsub. Central to this hierarchy is a dissemination guarantee: once a message is committed at its source, every eligible (non-faulty, active, and subscribed) peer eventually receives it. This property holds deterministically for Broadcastsub, Floodsub, and for Meshsub under identified network and parametric conditions. We show that this dissemination guarantee is violated for Gossipsub v1.1 using property-based testing and counter-example generation. The protocol admits executions in which committed messages are never delivered to subscribed peers. We use these counterexamples to construct concrete eclipse and network-partition attack gadgets. Simulation experiments confirm the feasibility of these attacks; following responsible disclosure, the vulnerability was assigned CVE-2022-47547

and acknowledged by the Gossipsub and Ethereum developers. These findings prompted protocol revisions and contributed to ongoing efforts to strengthen consensus-layer message dissemination in Ethereum and related systems.

Finally, we conduct a systematic exploration of the Gossipsub parameter space under network churn. Pareto analysis over delivery reliability and bandwidth cost yields three findings: (1) Gossipsub strictly dominates Floodsub across churn regimes; (2) minimal-gossip configurations lie on the Pareto frontier and dominate Ethereum's deployed parameters under worst-case delivery; and (3) gossip-based recovery, rather than eager-push redundancy, is the primary driver of reliable dissemination under churn. These results provide the first principled foundation for parameter selection in production Gossipsub networks.

Taken together, this work demonstrates that compositional refinement, combined with property-based testing, counter-example generation, and simulation based analyses, constitutes a practical and scalable methodology for analyzing complex, real-world P2P protocols. Beyond certifying correctness, it enables the discovery of previously unknown vulnerabilities and the principled optimization of deployed systems. All models, proofs, and simulation artifacts are publicly available and fully executable to support reproducibility and reuse.

Thesis supervisor: Panagiotis Manolios
Title: Professor

Acknowledgments

I like to think of my PhD journey as that of a growing child. I am deeply grateful to my advisor, Pete. Pete treats his students as his children, and like a father, he nurtured me with patience, trust, and care. Throughout my PhD, he consistently supported me in all aspects of my work and life: technical, administrative, financial, and personal. His optimism and enthusiasm are contagious, and they consistently fostered a supportive and motivating research environment. His confidence in me during moments of doubt, and his steady guidance during moments of growth, shaped not only my research but also who I am as a person. I could not have asked for a better mentor.

I would like to thank the members of my thesis committee: Pete Manolios, Cristina Nita-Rotaru, Joseph Kiniry and Guevara Noubir. Their feedback and guidance have greatly contributed to improving the quality of this dissertation.

Our research group felt like a family. Andrew Walter is my academic twin brother: we started our PhDs together under Pete during Fall 2018. Over the years, we worked side by side during internships at Amazon and Rivos Inc. and collaborated on multiple research projects. Some of my work directly builds on his contributions to ACL2s systems programming, Lisp-Z3 and ACL2s Docker infrastructure. Beyond academia, I was grateful to host Andrew at my wedding in December 2024. I cannot imagine this journey without him by my side.

Max von Hippel and I first met as collaborators on a group project for Cristina's course on computer network systems, where we studied the correctness of the Gossipsub scoring function. That project ultimately became the foundation of this dissertation. From the very beginning, I was astounded by Max's speed, rigor, and work ethic. Over the years, our collaboration extended well beyond research. We also lived together as roommates, and Max kindly invited me to be part of his bachelor party. Working with him pushed me to be more disciplined, to think more clearly, and to work more carefully. Our collaboration shaped not only the technical direction of this research, but also my standards for what good research should look like.

I was also fortunate to be part of a broader research group (extended family) around Pete that made graduate school both intellectually stimulating and genuinely enjoyable. I benefited greatly from working with and learning alongside Derek Egolf, Zeke Medley, Emmanuel Manolios, Angelina Manolios, Cassidy Waldrip, Shaoqi Wang, Ben Quiring, Brent (Yibo) Zhao, Joshua Wallin, Atharva Shukla, Ben Boskin, Konstantinos Athanasiou, and Justin Slepak. Whether through collaborations, classes, or informal discussions, each of them contributed to an environment that valued rigor, curiosity, and mutual support.

Students of formal methods working under professors Stavros Tripakis and Thomas Wahl

were also wonderful company and collaborators. I am especially grateful for interactions with Daniel Melcer, William Schultz, Lisa Oakley, Yuhao Zhou, Yijia Gu, and Peizun Liu. Beyond formal methods, I was equally fortunate to share my graduate years with a wider community of thoughtful and generous peers, including Girik Malik, Akshar Varma, Kashif Imteyaz, Baidik Chandra, Bryant Curto, Artem Pelenitsyan, Julia Belyakova, Sara Di Bartolomeo, Dustin Lin, Nathan Partlan, Daniel Patterson, Jethro Sun, Cameron Moy, Andrew Wagner, Ming-Ho Yee, and Lydia Zakynthinou. Their friendship, conversations, and support made Northeastern feel like a home, and my time here far richer than it would have been otherwise.

Just as it takes a village to raise a child, this journey would not have been possible without the support of many people and organizations. My two internships at Amazon with Jared Davis, Vaibhav Sharma, Andrew Gacek and Wolf Honoré were my first exposure to formal methods in an industrial setting, and significantly shaped how I think about applied research and real-world impact. I was fortunate to work with Mitesh Jain, a former student of Pete, at Rivos Inc. (now Meta), whose guidance and perspective on refinement proofs were invaluable. I sincerely thank Yiannis Psaras, Dimitris Vyzovitis, Nishant Das, Adrian Manning, Max Inden and others from ProbeLab for their thoughtful and in-depth discussions on Gossipsub, which greatly informed my understanding of the protocol's design. I also thank the Ethereum Foundation, especially Tyler Holmes, and to the PhD Dissertation Completion Fellowship for their financial support for portions of this dissertation. I thank the ACL2 community for being welcoming and supportive. I greatly enjoyed participating in the ACL2 Workshops since 2020, and look forward to attending future workshops.

I am deeply thankful to my family for providing an unwavering foundation throughout this journey. Their sacrifices, encouragement, and belief in me allowed me to pursue this path with confidence, even during its most uncertain phases. They gave me the freedom to explore, the strength to persevere, and the reassurance that I was never walking this path alone. I am especially thankful to my sister Shilpa for sending me rakhis every year.

I am equally grateful to my friends Chandra Shukla, Vivek Maithani, Rakshit Sharma and Arpit Agarwal, who have been with me since my formative years. Beyond academic support, they generously shared guidance on many non-academic matters, including navigating life in the United States and immigration-related challenges. I cherished the opportunities to visit them during short breaks while I was in the U.S., and their enduring friendship has been a constant source of comfort, stability, and joy throughout this journey.

Above all, I am profoundly grateful to my wife, Yoshita. She has been my constant source of strength, patience, and understanding. Through long nights, failed proofs, looming deadlines, and moments of self-doubt, she stood by me with unwavering care and belief, often from across continents. Her support carried me through this journey, and it would not have been possible without her.

As this PhD comes to an end, I see it as a child finally stepping into the world, shaped by the care, guidance, and love of many. While this dissertation marks the conclusion of one chapter, it also represents a beginning, carrying forward the lessons, values, and support of all those who helped raise it. For that, I am profoundly grateful.

Contents

List of Figures 15
List of Tables 17

I Introduction and Preliminaries **19**

1 Introduction **21**
1.1 Background and Context . . . 21
1.2 Gossipsub . . . 22
1.3 Problem Statement . . . 23
1.4 Approach: Compositional Refinement . . . 23
1.5 Contributions . . . 24
1.6 Thesis Statement . . . 24
1.7 Thesis Organization . . . 24

2 Preliminaries **27**
2.1 Notations . . . 27
2.2 Transition Systems . . . 29
2.3 Notions of Correctness . . . 29
2.4 Refinement . . . 32
2.5 Temporal Logic . . . 34
2.6 The ACL2s Theorem Prover . . . 35
2.7 Commitment Implies Delivery Property . . . 36

II Feasibility of Compositional Refinement **39**

3 Formalization and Verification of Floodsub **41**
3.1 Formal Description of the Broadcastsub model . . . 42
3.2 Formal Description of the Floodsub model . . . 44
3.3 Correctness Conditions for Floodsub . . . 47
3.4 Choosing a refinement map . . . 50
3.4.1 Choosing a Notion of Correctness . . . 51
3.5 The Refinement Proof . . . 52
3.6 Discussion on the Connectedness Condition . . . 67

4 Mechanized proof of Refinement of Floodsub **69**
4.1 Model Descriptions . . . 70
4.1.1 Broadcastnet . . . 70
4.1.2 Floodnet . . . 74
4.2 Correctness and the Refinement Theorem . . . 78
4.3 Proof Organization . . . 84
4.4 Bibliographic Notes . . . 85

5 Formalization and Verification of Meshsub **87**
5.1 Formal Description of the Meshsub Model . . . 88
5.2 Correctness Conditions for Meshsub . . . 94
5.3 Correctness of Meshsub . . . 95
5.3.1 The Refinement Proof . . . 95
5.3.2 Commitment Map . . . 96
5.4 Extension to Gossipsub . . . 103

6 Formalization and Property-Based Testing of Gossipsub v1.1 **105**
6.1 GossipSub ACL2s Model Description . . . 106
6.2 Reasoning about the scoring function . . . 115
6.3 Discussion: Relation to Floodsub and Meshsub Correctness Arguments . . . 118

III Effectiveness of the Combined Methodology **121**

7 Model based Security Analysis of Gossipsub v1.1 **123**
7.1 Attack Gadgets . . . 124
7.2 Eth2.0 Topologies . . . 125
7.3 Experimental Setup . . . 125
7.3.1 Eclipse Attacks . . . 127
7.3.2 Partition Attacks . . . 127
7.4 Discussion: Attacks as Refinement Counterexamples . . . 128

8 Lean Gossip–Simulation-Based Optimization of Gossipsub **129**
8.1 Methodology . . . 131
8.1.1 Gossipsub Overview . . . 131
8.1.2 Model Simplifications . . . 132
8.2 Protocol Implementation . . . 133
8.2.1 Aggregate Message Workload . . . 133
8.2.2 Network Model . . . 134
8.2.3 Simulation Timing . . . 135
8.2.4 Bandwidth Model . . . 135
8.2.5 Evaluation Metrics . . . 135
8.2.6 Pareto Dominance Analysis . . . 143
8.2.7 Simulation Parameters . . . 144
8.2.8 Validation Against Published Measurements . . . 144

8.3 Simulation Results . . . 148
8.3.1 Effect of Mesh Degree . . . 148
8.3.2 Effect of Gossip Degree . . . 151
8.3.3 Best-effort Trade-offs and Dominance . . . 151
8.3.4 Tradeoff Against Ethereum's Deployed Configuration . . . 153
8.4 Discussion . . . 153
8.4.1 Why Floodsub Lies Off the Pareto Frontier Under Churn . . . 153
8.4.2 Why Lean Gossip Strictly Dominates Ethereum's Configuration . . . 154
8.4.3 Recovery Dominates Redundancy . . . 154
8.4.4 Practical Recommendations . . . 154
8.5 Limitations and Future Work . . . 155
8.6 Bibliographic Notes . . . 156

IV Conclusion 159

9 Conclusion 161
9.1 Future Work . . . 162
9.2 Artifact Availability . . . 162

References 163

Vita 171

Index 171

List of Figures

3.1.1 Example of a message broadcast 1 43
3.3.1 `Floodsub` network illustration 47
3.3.2 Example message broadcast 2 47
3.3.3 Example of message dissemination failure 1 48
3.3.4 Example of message dissemination failure 2 48

4.2.1 Trace comparison: Floodnet vs Broadcastnet 80

5.3.1 Evolution of the message frontier $\mathcal{F}$ in `Meshsub` (red) and `Floodsub` (green). . 97

6.1.1 Topic mesh in a Gossipsub network 106
6.1.2 Eth2.02.0 community graph 107
6.2.1 Example Gossipsub network 115

7.1.1 Example AG_2 attack gadget 124
7.2.1 Ropsten topology visualization 126

8.3.1 Effect of Mesh Degree on delivery rate, bandwidth cost, and latency 149
8.3.2 Effect of Gossip Degree on delivery rate, bandwidth cost, and latency 150
8.3.3 Pareto dominance: delivery vs. normalized cost 152

List of Tables

3.1 →`Broadcastsub` definition. . . . 43
3.2 →`Floodsub` definition. . . . 46

5.1 →`Meshsub` definition, Part 1 . . . 90
5.2 →`Meshsub` definition, Part 2 . . . 92
5.3 Auxiliary `Meshsub` Transition Definitions . . . 93

6.1 Perturbations in topic-counters values violating Property 2 . . . 118

7.1 Eth2.0 Network Characteristics . . . 125
7.2 Wallclock time (mins) for attack simulation . . . 127

8.1 Ethereum message sizes . . . 133
8.2 Per-hop latency distributions (ms) by region pair . . . 134
8.3 Gossipsub Mesh Simulation Parameter Values . . . 145
8.4 Validation against Protocol Labs Gossipsub Evaluation . . . 146
8.5 Gossipsub configurations that dominate Floodsub . . . 151
8.6 Representative configurations across the reliability-efficiency-latency spectrum 155
8.7 Delivery rate by churn level for Gossipsub . . . 155
8.8 Diminishing returns of gossip degree . . . 156

Part I

Introduction and Preliminaries

Chapter 1

Introduction

Gossipsub is the message dissemination protocol underlying Ethereum, IPFS, and Filecoin—systems that collectively support billions of dollars in economic activity and a substantial fraction of today's decentralized Web3 infrastructure. Despite its central role, Gossipsub lacks a formal specification, its security mechanisms have been evaluated primarily through ad hoc testing and scenario-specific simulation, and its operational parameters were selected heuristically with limited quantitative justification. As a result, fundamental correctness, security guarantees and optimal configurations for these economically significant systems remain unclear.

At its core, any publish-subscribe (pubsub) protocol is expected to satisfy a fundamental dissemination property:

Once a message is committed at its source,

every eligible peer eventually receives it.

An *eligible* peer is one that is active (has not dropped or left), non-faulty (does not crash or deviate from the protocol specification) and *subscribed* to the topic of the message. Whether this property holds in modern, production-scale peer-to-peer (P2P) systems—and under what assumptions—has not been established formally.

This dissertation develops a compositional refinement-based methodology for rigorously analyzing P2P publish-subscribe protocols and applies it to Gossipsub. Our analysis reveals that Gossipsub v1.1 admits executions in which committed messages are never delivered to subscribed peers. We construct concrete eclipse and network partition attacks using attack gadgets that exploit this flaw; following responsible disclosure, the vulnerability was assigned MITRE CVE-2022-47547 and acknowledged by the Gossipsub and Ethereum developers. Beyond exposing this vulnerability, we demonstrate that principled refinement reasoning and systematic simulation enable both formal guarantees and practical parameter optimization for deployed P2P systems.

1.1 Background and Context

Peer-to-peer (P2P) systems are decentralized distributed systems in which nodes cooperate to form overlay networks over physical networks such as the Internet [2]. These systems are

inherently dynamic: nodes may join or leave at any time, and network membership may change rapidly due to churn. To achieve scalability and resilience, P2P protocols rely on self-organization and local decision-making.

Publish-subscribe (pubsub) systems constitute a widely studied class of P2P protocols that decouple information producers from consumers by routing messages according to logical topics. A publisher disseminates messages without direct knowledge of subscribers, while the network ensures delivery to interested peers. Early systems such as Scribe [8] implemented pubsub over structured overlays, demonstrating scalable topic-based dissemination.

A canonical and conceptually simple realization of pubsub is provided by *Floodsub* [73, 80]. In Floodsub, nodes maintain knowledge of their neighbors and forward published messages to all neighbors subscribed to the relevant topic. While simple and robust, naive flooding suffers from redundancy: messages may traverse multiple paths and be delivered multiple times, leading to bandwidth explosion. This phenomenon has been studied extensively, including in the context of the broadcast storm problem [66].

To mitigate redundancy, gossip-based (epidemic) dissemination protocols were developed. Seminal work by Demers *et al* [16] showed that limiting fanout to a logarithmic number of peers yields near-complete coverage within $\mathcal{O}(\log n)$ rounds with high probability under uniform random selection. Subsequent analytical and experimental studies [5, 20] demonstrated that periodic gossip with randomly selected peers enables scalable and resilient dissemination while significantly reducing bandwidth overhead. Modern pubsub systems combine eager message forwarding with gossip-based recovery to balance efficiency and robustness.

1.2 Gossipsub

Gossipsub v1.1 [47, 81] represents the state of the art in gossip-based pubsub for Web3 systems. For each topic, a peer maintains a bounded mesh of neighbors to which it eagerly pushes full messages. Peers outside this mesh receive lightweight gossip notifications containing message identifiers and may request the full message if needed. This hybrid push–pull approach reduces bandwidth overhead while preserving resilience.

Gossipsub introduces additional complexity through dynamic mesh management and scoring-based security mechanisms. Peers are added to or removed from topic meshes through grafting and pruning operations. Each peer maintains a local scoring function that evaluates neighbors based on behavioral metrics, including message delivery performance and protocol compliance. Scores influence mesh membership and are parameterized by application-specific weights. These mechanisms aim to mitigate spam, prevent eclipse attacks, and maintain robust connectivity under adversarial conditions.

While these features improve efficiency and security over naive flooding, they significantly increase protocol complexity. Message dissemination, control traffic, scoring updates, and mesh maintenance interact across multiple time scales. Global behavior emerges from local heuristics, making reasoning about end-to-end guarantees difficult. Existing evaluations rely primarily on testing and simulation; no formal specification captures the intended semantics of the protocol.

1.3 Problem Statement

Two fundamental challenges arise in modern gossip-based pubsub protocols.

First, correctness and security guarantees remain informal. In the absence of a formal specification, it is unclear whether the intended dissemination property holds, and under what assumptions. Security mechanisms are validated through empirical testing, leaving open the possibility of subtle vulnerabilities.

Second, parameter selection lacks principled justification. Gossipsub exposes numerous parameters controlling mesh degree, gossip fanout, timers, and scoring thresholds. These parameters interact in complex ways, influencing latency, bandwidth consumption, and delivery rate. Production deployments rely on heuristics rather than systematic analysis of trade-offs.

Addressing these challenges requires both formal reasoning and empirical evaluation.

1.4 Approach: Compositional Refinement

This dissertation introduces a compositional refinement framework for analyzing P2P pubsub protocols. We construct a hierarchy of protocol layers:

- **Broadcastsub**, an idealized protocol in which messages are delivered instantaneously to all subscribed peers in the network.
- **Floodsub**, which operates over an underlying peer connectivity graph, where messages are forwarded only along existing connections.
- **Meshsub**, which limits forwarding degree by constructing bounded-degree mesh overlays atop the underlying connectivity graph. Messages are forwarded only to mesh neighbors, while metadata about known messages may be gossiped to non-mesh neighbors.
- **Gossipsub v1.1**, which augments Meshsub with scoring-based mesh management to mitigate the influence of misbehaving peers.

In all of the above protocols, the underlying network is peer-to-peer (P2P), and peers are free to join or leave the system at arbitrary times. We prove that Floodsub refines Broadcastsub and that Meshsub refines Floodsub. By compositional refinement, Meshsub refines Broadcastsub. Within this hierarchy, we establish the central dissemination guarantee: once a message is committed at its source, every correct subscribed peer eventually receives it. This property holds deterministically for Broadcastsub and Floodsub, and with arbitrarily high probability for Meshsub under identified network conditions.

We then apply property-based testing and counterexample generation to Gossipsub v1.1. Our analysis demonstrates that the dissemination guarantee is violated: the protocol admits executions in which committed messages are never delivered to subscribed peers. From these counterexamples, we construct concrete eclipse and network partition attacks. Simulation experiments confirm their feasibility. Following responsible disclosure, the vulnerability was assigned CVE-2022-47547.

Finally, we conduct a systematic exploration of the Gossipsub parameter space under realistic churn. Using Pareto analysis over delivery reliability and bandwidth cost, we identify configurations that dominate Ethereum's deployed parameters and characterize the mechanisms driving reliable dissemination. We show that gossip-based recovery, rather than eager-push redundancy, is the dominant factor under churn.

1.5 Contributions

The principal contributions of this dissertation are:

1. The first formal, executable specification of Gossipsub, now recognized as the official reference model.
2. A compositional refinement hierarchy establishing formal dissemination guarantees for Floodsub and Meshsub.
3. The discovery and responsible disclosure of novel attacks against Gossipsub v1.1 (CVE-2022-47547).
4. A systematic parameter analysis demonstrating that production configurations are Pareto-dominated by leaner alternatives under realistic network conditions.
5. A practical methodology combining refinement proofs, property-based testing, counterexample generation, and simulation for analyzing complex, real-world P2P protocols.

1.6 Thesis Statement

In this thesis, we use formal methods, specifically compositional refinement proofs, property-based testing, counter-example generation and parameter-driven simulations for exploring (1) vulnerabilities and attacks against Gossipsub networks, and (2) the protocol parameter space to find optimum performance criteria. Following is our thesis statement.

Thesis statement: *A compositional refinement-based approach that synergistically combines formal verification, counterexample generation, and parameter-driven simulation is both feasible and effective for the analysis of peer-to-peer (P2P) protocols, enabling the discovery of novel attacks and the systematic improvement of protocol performance metrics.*

1.7 Thesis Organization

This dissertation is organized into four parts, each addressing a distinct stage of the compositional refinement-based analysis of gossip-based pubsub protocols.

Part I: Introduction and Preliminaries This part introduces the formal framework and core abstractions used throughout the thesis. Chapter 2 presents the modeling assumptions for peer-to-peer networks and the refinement notions employed in our analysis.

Part II: Feasibility of Compositional Refinement This part develops formal proofs of correctness for peer-to-peer publish–subscribe protocols using refinement-based verification. Chapter 3 defines Broadcastsub as an abstract dissemination specification and Floodsub as a flooding-based realization, and establishes that Floodsub refines Broadcastsub, yielding deterministic eventual-delivery guarantees under stated assumptions. Chapter 4 shows a mechanized proof of refinement of a simpler variant of Floodsub, where pending messages are stored as sets.

Chapter 5 introduces Meshsub (Gossipsub v1.0) and proves that `Meshsub` refines `Floodsub` via a commitment map `m2f` that abstracts away in-transit messages until they have left both the pending queue and the gossiping frontier. By the Compositional Skipping Refinement theorem, `Meshsub` therefore also refines `Broadcastsub` via `f2b` $\circ$ `m2f`. The refinement holds under three correctness conditions: overlay temporal connectedness condition *OverlayTConnectedp*, a positive gossip fanout ($D_{lazy} \geq 1$) and a non-empty recently-seen window ($W \geq 1$).

Part III: Effectiveness of the Combined Methodology This part analyzes Gossipsub v1.1, focusing on its scoring-based mesh management and security mechanisms. Chapter 6 presents the first formal, executable specification of Gossipsub and applies property-based testing and counterexample generation to the model. In chapter 7, we demonstrate violations of the dissemination guarantee, construct concrete eclipse and network partition attacks, and describe their responsible disclosure as CVE-2022-47547. Chapter 8 explores the Gossipsub parameter space under realistic churn through systematic simulation and Pareto analysis over delivery reliability and bandwidth cost. We identify configurations that dominate production deployments and characterize the mechanisms driving robust dissemination.

Part IV: Conclusion Chapter 9 summarizes the results and discusses broader implications, limitations, and directions for future work.

All mechanized proofs, counterexample artifacts, and simulation infrastructure are publicly available at the repository referenced in Section 9.2.

Chapter 2

Preliminaries

In this chapter, we introduce the notation used throughout this dissertation and review the notions of correctness and refinement relevant to our work.

2.1 Notations

$\mathbb{N}$ denotes the set of natural numbers $\{0, 1, 2, \ldots\}$. ω is the first infinite ordinal. We denote tuples using angled brackets; for example, the tuple consisting of elements $a_1, a_2, \ldots, a_n$ is written as $\langle a_1, a_2, \ldots, a_n \rangle$. We denote sequences using square brackets; for example, the sequence consisting of elements $a_1, a_2, \ldots, a_n$ is written as $[a_1, a_2, \ldots, a_n]$. The empty sequence is denoted $[\,]$.

We write $f(x)$ to denote the value of a function f applied to an argument x. The expression $\#S$ denotes the cardinality of a set S.

We abbreviate the update $S := S \cup P$ by $S :\cup\!= P$. Similarly, we abbreviate $S := S \setminus P$ by $S :\setminus\!= P$. Given a record $r = \langle \ldots, \mathit{field}, \ldots \rangle$, we denote the value of the *field* component of r by $r.\mathit{field}$. The operator :: denotes list cons and is used to add an element to the beginning of a list.

We use dot notation to invoke functions on structured values. For a tuple $t = \langle x, y \rangle$, we write $t.\underline{\text{first}} = x$ and $t.\underline{\text{second}} = y$.

For a sequence or list l, we write $l.\underline{\text{head}}$ and $l.\underline{\text{last}}$ to denote its first and last elements, respectively. We write $l.\underline{\text{tail}}$ for the list obtained by removing the first element, and $l.\underline{\text{init}}$ for the list obtained by removing the last element. For a predicate p, the expression $l.\underline{\text{filter}}(p)$ denotes the subsequence of l consisting of elements that satisfy p.

For a queue q, we write $q.\underline{\text{enqueue}}(x)$ to denote the queue obtained by enqueuing element x into q, and $q.\underline{\text{dequeue}}$ to denote the queue obtained by dequeuing the front element of q. We write $q.\underline{\text{front}}$ to denote the element at the front of q.

The function $\underline{\text{count}}(s, x)$ denotes the number of occurrences of element x in the sequence s. The operators $\underline{\text{extend}}$ and $\underline{\text{delete}}$ denote extension and deletion operations, whose precise behavior will be clear from context. For a function f, $\mathsf{dom}(f)$ denotes its domain and $\mathsf{rng}(f)$ denotes its range. For a partial function f and a key k, $f.\mathsf{removeKey}(k)$ denotes the partial function obtained by removing k from the domain of f:

$$f.\mathsf{removeKey}(k) = \langle \lambda x \in \mathsf{dom}(f) \setminus \{k\} :: f(x) \rangle.$$

as the set of keys with non-empty values. For a total function f mapping to sets, we define $\mathsf{dom}^+(f) := \{x \in \mathsf{dom}(f) \mid f(x) \neq \varnothing\}$

We use standard conditional expressions of the form $\underline{\text{if}}\ G\ \underline{\text{then}}\ A\ \underline{\text{else}}\ B$, and write $\underline{\text{otherwise}}$ for the default case in piecewise definitions. The construct $\underline{\text{let}}\ x = e\ \underline{\text{in}}\ b$ denotes let-binding, and $\underline{\text{skip}}$ denotes the identity operation.

An expression of the form $\langle Qx : r : b\rangle$ denotes a quantified expression, where Q is a quantifier, x is the bound variable, r is the range predicate for x (taken to be $\underline{\textit{true}}$ if omitted), and b is the body. We sometimes write $\langle Qx \in X : r : b\rangle$ as an abbreviation for $\langle Qx : x \in X \wedge r : b\rangle$, where, as before, r is $\underline{\textit{true}}$ if omitted; we may further abbreviate $\langle Qx \in X : \underline{\textit{true}} : b\rangle$ as $\langle Qx \in X :: b\rangle$, placing the range directly in the first slot and omitting the range predicate. When b is a rule, we interpret $\langle \forall x : r : b\rangle$ as the sequential composition of all instantiated rules $b[x]$ over values of x satisfying r.

For any binary relation R, we abbreviate $\langle s, w\rangle \in R$ by sRw. For a set S, we write $R(S)$ for the image of S under R, *i.e.*,

$$R(S) = \{\, y \mid \exists x \in S : xRy \,\}.$$

The composition of binary relations R and T is denoted $T \circ R$ and is defined by

$$T \circ R = \{\, \langle r, t\rangle \mid \exists x : rRx \wedge xTt \,\}.$$

The inverse of a binary relation R, denoted R^{-1}, is defined as

$$R^{-1} = \{\, \langle a, b\rangle \mid bRa \,\}.$$

Let $B \subseteq X \times X$ be a binary relation. The relation B is *reflexive* if $\langle \forall x \in X : xBx\rangle$. It is *symmetric* if $\langle \forall x, y \in X : xBy \Rightarrow yBx\rangle$. It is *antisymmetric* if $\langle \forall x, y \in X : xBy \wedge yBx \Rightarrow x = y\rangle$. It is *transitive* if $\langle \forall x, y, z \in X : xBy \wedge yBz \Rightarrow xBz\rangle$.

A binary relation is a *preorder* if it is reflexive and transitive. A preorder that is also symmetric is an *equivalence relation*. A preorder that is antisymmetric is a *partial order*.

If $\leq_1$ is a partial order on X, $\leq_2$ is a partial order on Y, and $f : X \to Y$, then f is *monotonic* if

$$a \leq_1 b \Rightarrow f(a) \leq_2 f(b)$$

for all $a, b \in X$.

A well-founded structure is a pair $\langle W, \lessdot\rangle$, where W is a set and $\lessdot$ is a binary relation on W such that there are no infinitely decreasing sequences in W with respect to $\lessdot$. We use $<$ to compare natural numbers and $\prec$ to compare ordinal numbers.

From highest to lowest binding power, the operators are ordered as follows: parentheses; function application; binary relations (*e.g.*, sBw); equality ($=$) and set membership ($\in$); conjunction ($\wedge$) and disjunction ($\vee$); implication ($\Rightarrow$); and finally, binary equivalence ($\equiv$). Whitespace is used to reinforce binding strength. Greater spacing indicates lower binding power. Unless parentheses are used, expressions are parsed according to this precedence order.

2.2 Transition Systems

Definition 2.2.1 (Labeled Transition System (LTS))
A labeled transition system (LTS) is a structure $\langle S, \rightarrow, L \rangle$, where S is a non-empty set of states, $\rightarrow \subseteq S \times S$ is a left-total transition relation, and L is a labeling function whose domain is S.
A path is a sequence of states such that for adjacent states s and u, $s \rightarrow u$. A path σ is a *fullpath* if it is infinite. $fp(\sigma, s)$ denotes that σ is a fullpath starting at s, and for $i \in \omega$, $\sigma(i)$ denotes the i-th element of path σ.

We will often define transitions in an LTS using rules. We treat rules as functions from states to states. A rule of the form $G \rightarrow A$ denotes the function $\langle \lambda s :: \textbf{if } G \textbf{ then } A \textbf{ else } s \rangle$, where s may occur freely in both G and A. If A itself is used as a rule, we regard it as an abbreviation for $\langle \lambda s :: \underline{true} \rightarrow A \rangle$, *i.e.*, the rule that always applies A. For sequencing transitions, we write $A \fatsemi B$ to denote the sequential composition of rules. Formally,

$$(A \fatsemi B)(s) = B(A(s)).$$

2.3 Notions of Correctness

A notion of correctness specifies what it means for an implementation to faithfully realize its specification. Such notions can be understood as constraints on observable behavior: an implementation is correct exactly when all of its behaviors are permitted by the specification. A useful notion of correctness must be strong enough to rule out obviously incorrect implementations, and permissive enough to accommodate for differences in observed behaviour arising from differing levels of abstraction. We define the following notions of correctness used throughout this dissertation.

Definition 2.3.2 (Simulation)
A relation R is a *simulation relation* [64] on a transition system $M = \langle S, \rightarrow, L \rangle$ if $R \subseteq S \times S$ and for all $s, w \in S$ such that $s \, R \, w$, the following conditions hold:

1. $L(s) = L(w)$;
2. for all $u \in S$, if $s \rightarrow u$, then there exists $v \in S$ such that $w \rightarrow v$ and $u \, R \, v$.

Definition 2.3.3 (Bisimulation)
A relation B is a *bisimulation relation* [65, 68] on a transition system $M = \langle S, \to, L \rangle$ if $B \subseteq S \times S$ and for all $s, w \in S$ such that $s\, B\, w$, the following conditions hold:

1. $L(s) = L(w)$;
2. for all $u \in S$, if $s \to u$, then there exists $v \in S$ such that $w \to v$ and $u\, B\, v$;
3. for all $v \in S$, if $w \to v$, then there exists $u \in S$ such that $s \to u$ and $u\, B\, v$.

We say that s is *similar* to w if there exists a simulation relation R such that $s\, R\, w$. We say that s is *bisimilar* to w if there exists a bisimulation relation B such that $s\, B\, w$. If s is similar to w and w is similar to s, then it does not imply that s is bisimilar to w [69].

Stuttering simulation [55] allows an implementation to take several internal steps while remaining consistent with a single step of the specification, provided that observable behavior is preserved. This notion is intended to capture differences in granularity between abstract and concrete models, where internal behavior of the implementation may not have a direct counterpart at the specification level. Proving a stuttering simulation will require reasoning about infinite sequences, which is not amenable to mechanical verification. For mechanized proofs, it is essential to have localized proof rules. This brings us to Well Founded Simulation, which provides a complete characterization of stuttering simulation.

Definition 2.3.4 (Well-Founded Simulation (WFS) [55])
$B \subseteq S \times S$ is a well-founded simulation on LTS $M = \langle S, \to, L \rangle$ iff:

- **WFS1.** $\langle \forall s, w \in S :: sBw \Rightarrow L(s) = L(w) \rangle$, and
- **WFS2.** There exists functions, $rankt : S \to W$, $rankl : S \times S \to \mathbb{N}$, such that $\langle W, \prec \rangle$ is well-founded, and $\langle \forall s, u, w \in S :: sBw \wedge s \to u \Rightarrow$
  - $\langle \exists v :: w \to v \wedge uBv \rangle \quad \vee$
  - $(uBw \wedge rankt(u, w) \prec rankt(s, w)) \quad \vee$
  - $\langle \exists v :: w \to v \Rightarrow (sBv \wedge rankl(v, s, u) < rankl(w, s, u)) \rangle$

We sometimes require a stronger notion: equivalence bisimulation, which demands that related states can match each other's transitions symmetrically. Well-Founded Equivalence Bisimulation (WEB) provides a mechanizable proof rule for stuttering equivalence.

Definition 2.3.5 (Well-Founded Equivalence Bisimulation (WEB) [55])
$B \subseteq S \times S$ is a well-founded equivalence bisimulation on LTS $M = \langle S, \rightarrow, L\rangle$ iff:

- **WEB1.** B is an equivalence relation on S; and
- **WEB2.** $\langle \forall s, w \in S :: sBw \Rightarrow L(s) = L(w)\rangle$; and
- **WEB3.** There exist functions $rankt : S \rightarrow W$, $rankl : S \times S \rightarrow \mathbb{N}$, such that $\langle W, \lessdot\rangle$ is well-founded, and $\langle \forall s, u, w \in S :: sBw \wedge s \rightarrow u \Rightarrow$
  - (a) $\langle \exists v :: w \rightarrow v \wedge uBv\rangle \quad \vee$
  - (b) $(uBw \wedge rankt(u) \lessdot rankt(s)) \quad \vee$
  - (c) $\langle \exists v :: w \rightarrow v \wedge sBv \wedge rankl(v, u) < rankl(w, u)\rangle$

The key difference from WFS is condition WEB1: the relation B must be an equivalence (reflexive, symmetric, and transitive), ensuring that the refinement is bidirectional.

Skipping simulation is a notion of correctness that characterizes when an implementation faithfully realizes a specification, while allowing the implementation to advance more rapidly than the specification [26]. Skipping can be thought of as the dual of stuttering: while stuttering allows us to *stretch* executions of the specification to match the implementation, skipping allows us to *squeeze* them. This is necessary when reasoning about optimized systems that execute faster than their specifications — systems in which a single concrete step may perform the work of multiple abstract steps.

Definition 2.3.6 (Smatch)
Let INC be the set of strictly increasing sequences of natural numbers starting at 0. Given a fullpath σ, the i-th segment of σ with respect to $\pi \in INC$, written $\pi\sigma_i$, is given by the sequence $\langle \sigma(\pi.i), \ldots, \sigma(\pi.(i+1) - 1)\rangle$. For $\pi, \xi \in INC$ and relation B, we define:

$$scorr(B, \sigma, \pi, \delta, \xi) \equiv \langle \forall i \in \omega :: \langle \forall s \in {}^{\pi}\sigma^{i} :: s\, B\, \delta(\xi.i)\rangle\rangle$$

and

$$smatch(B, \sigma, \delta) \equiv \langle \exists \pi, \xi \in INC :: corr(B, \sigma, \pi, \delta, \xi)\rangle$$

Definition 2.3.7 (Skipping Simulation (SKS))
$B \subseteq S \times S$ is a skipping simulation (SKS) on transition system $M = \langle S, \rightarrow, L\rangle$ iff for all $s, w \in S$ such that sBw, the following hold:

- **SKS1.** $L(s) = L(w)$; and
- **SKS2.** $\langle \forall \sigma :: fp(\sigma, s)\rangle\langle \exists \delta :: fp(\delta, w)\rangle smatch(B, \sigma, \delta)$

where $fp(\sigma, s)$ denotes that σ is a fullpath starting from s.

An equivalent notion of correctness is Reduced well-founded skipping which is amenable to mechanical verification as it does not have to deal with infinite paths.

Definition 2.3.8 (Reduced Well-Founded Skipping (RWFSK) [26])
$B \subseteq S \times S$ is a well-founded skipping relation on LTS $M = \langle S, \rightarrow, L \rangle$ iff:

- **RWFSK1.** $\langle \forall s, w \in S :: sBw \Rightarrow L.s = L.w \rangle$, and
- **RWFSK2.** There exists a function, rankt : $S \times S \rightarrow W$, such that $\langle W, \prec \rangle$ is well-founded, and $\langle \forall s, u, w \in S :: sBw \wedge s \rightarrow u \Rightarrow$
  - $(uBw \wedge rankt(u, w) \prec rankt(s, w)) \vee$
  - $\langle \exists v :: w \rightarrow^+ v \wedge uBv \rangle$

where $\rightarrow^+$ denotes the transitive closure of the transition relation.

2.4 Refinement

Refinement [1] is a specification method which involves showing that any infinite behavior of a system is related to infinite behaviors of another high-level abstract system that serves as its specification.

Consider a fixed and connected network of n peers $P = \{1, \ldots, n\}$, all subscribed to topic t. For simplicity, we disallow peers from joining or leaving: the peer set is fixed. We consider two protocol abstractions, which we call Protocol F and Protocol B.

Protocol F is the concrete system: each peer state holds a `Pdg` (pending queue) and a `Seen` set. Peer 1 first produces m into its pending queue via `produce`, then forwards m to each neighbor one at a time via `forward`, moving m from `Pdg` to `Seen` as each step completes. Since the network is connected, m eventually reaches all n peers through a sequence of forwarding steps.

Protocol B is the abstract system: each peer state holds only a `Seen` set. The single transition `bcast` atomically delivers m to every peer in one step.

Refinement Maps We use a Refinement Map to map Protocol F states to "related" (based on the chosen notion of correctness) Protocol B states because these protocols are at different levels of abstraction. Notice that they represent data differently: Protocol B peer states do not have `Pdg` queues. Using a refinement map is like putting on glasses that allow us to "see" lower-level concrete states as their abstract specification states.

Two refinement maps for the same pair of protocols. There are two natural ways to map Protocol F states to Protocol B states, and the choice determines which notion of refinement applies.

Map r_1 (pending suppression). r_1 suppresses pending messages: $(r_1(s))(i).\mathsf{Seen} = s(i).\mathsf{Seen} \setminus \mathcal{P}(s)$, where $\mathcal{P}(s)$ is the set of messages still pending in any peer's `Pdg`. Only fully committed messages appear in the abstract image.

Map r_2 (projection). r_2 simply projects `Seen` sets: $(r_2(s))(i).\mathsf{Seen} = s(i).\mathsf{Seen}$, with no suppression of pending messages.

Under r_1, every produce and forward step in Protocol F maps to the same Protocol B state, as long as m remains pending in some peer's Pdg: r_1 suppresses m and the abstract state does not change. Only after the final forward, when m has left all pending queues, does the abstract image advance to match the single bcast step in Protocol B.

$$\underbrace{s_0 \xrightarrow{\text{produce}} s_1 \xrightarrow{\text{forward}} s_2 \cdots \xrightarrow{\text{forward}} s_k}_{k \text{ concrete steps}} \qquad \underbrace{w_0 \xrightarrow{\text{bcast}} w_1}_{1 \text{ abstract step}}$$

This is *stuttering*: states s_0 through s_{k-1} all map to w_0 under r_1, and only s_k maps to w_1. A well-founded rank — the number of peers yet to receive m — decreases at each step, ensuring the stuttering is finite. Crucially, the refinement relation holds at *every* intermediate concrete state.

Now, consider a different refinement map: r_2, which simply projects the Seen sets. Under r_2, consider the state s_j reached after peer 1 has forwarded m to some but not all peers. For concreteness, suppose after j forwarding steps, peers $\{1, \ldots, j\}$ have m in their Seen sets but peers $\{j{+}1, \ldots, n\}$ do not. Then $(r_2(s_j))(i).\mathsf{Seen} = \{m\}$ for $i \leq j$ and $\varnothing$ otherwise.

This partial delivery state has *no counterpart in Protocol B*. Protocol B has only two states for message m: either no peer has seen m (before bcast), or all peers have seen m (after bcast). There is no Protocol B state where some peers have m and others do not.

Stuttering simulation cannot handle this: it requires the refinement relation to hold at every intermediate concrete state s_j, but $r_2(s_j)$ does not correspond to any reachable Protocol B state. Skipping refinement relaxes this: it only requires the relation to hold at states where both systems have made matching observable progress — concretely, at s_0 (before any delivery) and s_k (after all deliveries) — allowing the intermediate states $s_1, \ldots, s_{k-1}$ to have no abstract counterpart.

Here s_0 relates to w_0 and s_k relates to w_1, but $s_1, \ldots, s_{k-1}$ — where delivery is partial — have no abstract counterpart in Protocol B. Skipping refinement permits exactly this: the concrete path from s_0 to s_k matches the abstract bcast step at the endpoints, without requiring every intermediate state to be related.

Having motivated both notions, and having defined the background for relating states for correctness, we now apply these notions to transition systems. In the following definitions, M_A can be considered as a specification while M_C can be considered its concrete implementation.

Definition 2.4.9 (Simulation Refinement [55])

Let $M_A = \langle S_A, \xrightarrow{A}, L_A \rangle$, $M_C = \langle S_C, \xrightarrow{C}, L_C \rangle$, and let $r : S_C \to S_A$ be a refinement map. We say that M_C is a simulation refinement of M_A with respect to r, written $M_C \sqsubseteq_r M_A$, if there exists a relation, B, such that all of the following hold.

1. $\langle \forall s \in S_A :: sB(r(s)) \rangle$ and
2. B is a WFS on the LTS $\langle S_A \uplus S_C, \xrightarrow{C} \uplus \xrightarrow{A}, \mathcal{L} \rangle$, where $\mathcal{L}(s) = L_C(s)$ if $s \in S_C$ else $\mathcal{L}(s) = L_A(r(s))$.

Definition 2.4.10 (WEB Refinement [55])
Let $M = \langle S, \to, L \rangle$, $M' = \langle S', \to', L' \rangle$, and $r : S \to S'$. We say that M is a WEB refinement of M' with respect to refinement map r, written $M \approx_r M'$, if there exists a relation B such that $\langle \forall s \in S :: sB(r(s)) \rangle$ and B is a WEB on the transition system $\langle S \uplus S', \to \uplus \to', \hat{L} \rangle$, where $\hat{L}(s) = L'(s)$ for $s \in S'$ and $\hat{L}(s) = L'(r(s))$ otherwise.

Definition 2.4.11 (Skipping Refinement [26])
Let $M_A = \langle S_A, \xrightarrow{A}, L_A \rangle$, $M_C = \langle S_C, \xrightarrow{C}, L_C \rangle$, and let $r : S_C \to S_A$ be a refinement map. We say that M_C is a skipping refinement of M_A with respect to r, written $M_C \lesssim_r M_A$, if there exists a relation $B \subseteq S_C \times S_A$, such that all of the following hold.

1. $\langle \forall s \in S_C :: sB(r(s)) \rangle$ and
2. B is a RWFSK on the LTS $\langle S_C \uplus S_A, \xrightarrow{C} \uplus \xrightarrow{A}, \mathcal{L} \rangle$, where $\mathcal{L}(s) = L_A(s)$ if $s \in S_A$ else $\mathcal{L}(s) = L_A(r(s))$.

A key benefit of refinement-based correctness notions is compositionality, which supports modular reasoning about complex systems. By proving refinement between successive abstraction levels, one can compose these results to obtain end-to-end correctness. The following theorems capture this property for simulation refinement and skipping refinement.

Theorem 2.4.1 (Compositional Simulation Refinement [55]). *If* $M_1 \sqsubseteq_p M_2$ *and* $M_2 \sqsubseteq_r M_3$, *then*

$$M_1 \sqsubseteq_{r \circ p} M_3.$$

Theorem 2.4.2 (Compositional Skipping Refinement [28]). *If* $M_1 \lesssim_p M_2$ *and* $M_2 \lesssim_r M_3$, *then*

$$M_1 \lesssim_{r \circ p} M_3.$$

2.5 Temporal Logic

We use CTL* (Computational Tree Logic) to express temporal properties over execution traces.

Definition 2.5.12 (CTL* Operators)
The basic CTL* temporal operators are:

- $\mathsf{X}\,\phi$ - "neXt": ϕ holds in the next state
- $\mathsf{F}\,\phi$ - "Finally": ϕ holds eventually
- $\mathsf{G}\,\phi$ - "Globally": ϕ holds in all future states
- $\phi\ \mathsf{U}\ \psi$ - "strong Until": ϕ holds at every state strictly before the first state where ψ holds, and ψ must hold at some point; formally, there exists $k \geq 0$ such that ψ holds at position k and ϕ holds at all positions $0 \leq j < k$. Note that $\mathsf{F}\,\psi$ is equivalent to $\underline{\mathit{true}}\ \mathsf{U}\ \psi$.
- $\phi\ \mathsf{W}\ \psi$ - "Weak until": like strong until, but ψ is not required to ever hold; either ψ holds at some point with ϕ holding at all prior positions (as in strong until), or ϕ holds globally forever. Formally, either there exists $k \geq 0$ such that ψ holds at position k and ϕ holds at all $0 \leq j < k$, or ϕ holds at all positions. Note that $\mathsf{G}\,\phi$ is equivalent to $\phi\ \mathsf{W}\ \underline{\mathit{false}}$.

Path quantifiers combine with temporal operators:

- A - "for All paths"
- E - "there Exists a path"

Common combinations include AG (always globally), AF (always eventually), and AX (always next).

We write $ACTL^* \setminus X$ to denote the fragment of CTL* with universal path quantification, excluding the next-time operator X.

2.6 The ACL2s Theorem Prover

The ACL2 system, due to Kaufmann and Moore [30, 32, 33], consists of a programming language, a first-order logic, and a theorem prover. ACL2 is an abbreviation for *A Computational Logic* for *Applicative Common Lisp*.

The term "Computational Logic" refers to ACL2, the logic, which is a first-order logic. The logic can be extended via events, such as by proving new theorems, or by defining new functions. Since the unconstrained introduction of axioms can render the logic unsound, ACL2 has a definitional principle which restricts the functions one can define. The logic also includes rules of inference, including an induction principle that is used to reason about recursively defined functions. The theorem prover, described in the book [33], is used to check proofs mechanically using previously defined axioms, rules of inference and induction principle.

The term "Applicative Common Lisp" refers to ACL2, the language based on Lisp, which is a applicative, or purely functional programming language. This is precisely the principle of

referential transparency: equals may be substituted for equals in any context. Consequently, ACL2 functions are free of side effects in the logic, which simplifies reasoning.

ACL2 has a steep learning curve; for a new user, mastering the system can resemble a learner driver attempting to control a Formula One race car. The "ACL2 Sedan" (ACL2s) saves new users from common mistakes and provides powerful and expressive macros like `match`, helpful syntax checking, termination analysis, and testing facilities. Developed by Peter Dillinger, Pete Manolios, Harsh Chamarthi and Andrew Walter at Northeastern University, Boston, ACL2s is an extension of the ACL2 theorem prover.

In addition to the capabilities of ACL2, ACL2s provides:

(1) A powerful type system via the `defdata` data definition framework [13] and the `definec` and `property` forms, which support typed definitions and properties.

(2) Counterexample generation through the `cgen` framework, which synergistically integrates theorem proving, type reasoning, and testing [10–12].

(3) Advanced termination analysis based on calling-context graphs (CCGs) [62] and ordinals [59–61].

(4) An optional Eclipse IDE plugin [9].

(5) The ACL2s systems programming framework (ASPF) [84], which enables the development of Common Lisp tools that use ACL2, ACL2s, and Z3 as a service [37, 85–87].

2.7 Commitment Implies Delivery Property

A peer's ability to receive a message depends on three concurrent conditions.

Active. For a peer i in network state s (which maps peer ids to their states), we track whether it is active in s using the following predicate:

$$activep(i, s) = i \in \mathsf{dom}(s) \wedge s(i).\texttt{Act}$$

where `Act` is a flag in the peer state, which a leaving peer sets to *false*, and a rejoining peer sets to *true*.

Non-faulty. A peer is *non-faulty* if it is operating according to protocol semantics. A faulty peer—one that has crashed, or is otherwise unable to process transitions—is excluded from the delivery guarantee. For the P2P systems `Broadcastsub`, `Floodsub` and `Meshsub` considered in chapters 3 through 5, we assume that peers are not faulty.

Subscribed. Dissemination in pubsub systems is topic-scoped: a peer receives only messages whose topic it has explicitly subscribed to. The `Subs` field records the set of topics to which peer i is currently subscribed, and delivery is only guaranteed while $m.\texttt{Topic} \in s(i).\texttt{Subs}$ holds.

Together these three conditions define the notion of an *eligible* peer: one for which the protocol is both obligated and able to ensure delivery.

Definition 2.7.13 (Eligible Peer)

Let s be a global state with non-faulty peers, i a peer, and m a message. Peer i is *eligible for m in state s*, written $\mathsf{eligible}(i, m, s)$, if

$$\mathit{activep}(i, s) \ \wedge \ m.\texttt{Topic} \in s(i).\texttt{Subs}$$

When used inside temporal formulas, $\mathsf{eligible}(i, m)$ denotes evaluation of $\mathsf{eligible}(i, m, s)$ at the current state along the path.

We define a uniform liveness property relating commitment of a message at its origin to delivery to eligible peers.

Definition 2.7.14 (Commitment⇒Delivery Property)

The *Commitment⇒Delivery property* holds for a fullpath σ of P2P pubsub system $\mathcal{M}$ if for every message m, its originating peer $o = m.\texttt{Origin}$ and every peer i, the following LTL formula holds. Let $c(t) = m \in \sigma(t)(o).\texttt{Seen}$, $e(t) = \mathsf{eligible}(i, m, \sigma(t))$ and $d(t) = m \in \sigma(t)(i).\texttt{Seen}$ in

$$\mathsf{AG}\Big(\big(\neg c \ \mathsf{U} \ (c \wedge e)\big) \implies \Big(\neg c \ \mathsf{U} \ \Big(c \wedge \big(\mathsf{F}(d) \vee \mathsf{F}(\neg\, e)\big)\Big)\Big)\Big)$$

The antecedent captures the unique moment at which m is committed by its originator: m was absent from o's seen set at all prior states, and it is now present. If at this same moment peer i is also eligible to receive m, then the consequent guarantees that the system is in a state where o has not yet seen m, until the moment of commitment, after which one of two outcomes must eventually occur: either peer i receives m, or peer i becomes ineligible by departing, becoming faulty, or unsubscribing.

Whether the Commitment⇒Delivery property holds for a given protocol depends on the structure of the underlying network. In later chapters we will see that establishing this property for Floodsub and Meshsub requires a *temporal connectedness* condition: every eligible peer must lie on some path from the originator through which m can be forwarded, now or in the future.

Importantly, this condition cannot be checked locally by any single peer. A peer can only observe its own state and connection to neighbors; it has no global view of whether a path to every subscriber exists in the network. For example, consider a network where peer 1 originates m and peer 3 is subscribed, but the only path from peer 1 to peer 3 passes through peer 2. If peer 2 departs before forwarding m, the path is broken and peer 3 will never receive m — yet neither peer 1 nor peer 3 can determine locally whether peer 2 had already forwarded m before departing.

The temporal connectedness condition rules out such scenarios at the level of the protocol specification rather than at runtime. A full treatment of this condition and its role in the correctness proofs is given in later chapters; we do not discuss it further here.

Part II

Feasibility of Compositional Refinement

Chapter 3

Formalization and Verification of Floodsub

Floodsub is a simple, robust and popular P2P pubsub protocol, where nodes can arbitrarily leave or join the network, subscribe to or unsubscribe from topics and forward newly received messages. In this chapter, we introduce a refinement-based approach that formally relates a realistic flooding P2P protocol, Floodsub, to a simpler abstraction, Broadcastsub.

Our specification for Broadcastsub (`Broadcastsub`) comes from the description of a Publish/Subscribe system in the libp2p specification [73], modulo implementation details like peer discovery, connection and topology, and types of peering. Some of the salient features of `Broadcastsub` are:

- **Publish/Subscribe:** All messages are classified into topics. Peers can subscribe to or unsubscribe from topics.
- **Messaging:** Peers can publish messages in topics. Each message gets delivered to all peers subscribed to the topic. Message transmission is instantaneous *i.e.*, subscribers receive messages they subscribe to at the same time.
- **Open Network:** Peers are free to leave or join the network. We choose to have peers leaving the network forget their state. Hence, a peer rejoining a network is identical to a completely new peer.

Broadcastsub is a very simple P2P protocol, in fact, it is so simple that it is not even be possible to implement it in the real world, where we need to consider the maximum speed of message transmission across a medium, the network topology, etc. Then what is the point in defining it? The point of having a highly abstract specification like `Broadcastsub`, is that it allows us to state and prove properties about `Floodsub` with ease. For example, if we want to prove a property that every subscribing peer of a message that is in the `Broadcastsub` receives that message, we can prove it simply by referring to the definition of a `Broadcastsub`. For the lower levels of specifications, we can prove the same property given that a refinement map from those specifications to `Broadcastsub` exists, under the conditions required for the refinement map to exist.

We prove that Floodsub implements (refines) Broadcastsub and, consequently, that a class of safety and liveness properties established for Broadcastsub also hold for Floodsub,

provided a temporal network connectedness condition holds. To our knowledge, this is the first formal result transferring correctness guarantees via refinement over a dynamic network protocol where nodes are free to leave or join. In our work, we generalize the idea of nodes having adequate fanout by expressing the required connectedness of the network as a temporal connectedness condition: the graph induced by message delivery transitions must remain connected over time, even if individual message propagations steps are lossy, or the path along which the message is transmitted is disconnected. This assumption captures classical static connectivity assumptions, extends it to dynamic networks with node churn, and more importantly, satisfies known sufficient conditions from prior work. In particular, it implies the delivery path property used by Birman et al. [5] to analyze probabilistic broadcast, and is consistent with the path redundancy assumptions in Gossipsub. By phrasing this as a temporal predicate over event traces, our refinement-based formulation factors out probabilistic analysis, however we deal with the probabilistic aspects of message dissemination in Chapters 5 and 8.

This chapter is organized as follows. Sections 3.1 and 3.2 formalize the Broadcastsub model and the Floodsub model respectively. Section 3.3 states correctness conditions. Section 3.4 explains our choice of refinement map. Section 3.5 develops the refinement proof, including probabilistic analysis of gossip rounds. Section 3.6 concludes with a discussion of the connectedness condition required for our refinement proof.

3.1 Formal Description of the Broadcastsub model

Topics are elements of a set $\mathcal{T}$, and message payloads are strings. A message m is a tuple $\langle \texttt{Payload}, \texttt{Topic}, \texttt{Origin} \rangle$, where $m.\texttt{Payload}$ is the payload, $m.\texttt{Topic} \in \mathcal{T}$ is the topic under which the message is published, and $m.\texttt{Origin}$ is the peer where m originated from. We assume that all messages produced are globally unique: no two distinct production events yield the same message tuple. We could make this explicit by including a per-peer monotone counter in the payload, but we treat uniqueness as an axiom and do not model the counter explicitly.

We describe a `Broadcastsub` by its state which is a partial map from peers to their states. Each peer is identified by a distinct natural number from $\mathbb{N}$ (the set of natural numbers). Let s be a `Broadcastsub` state. The state for peer $i \in \texttt{dom}(s)$ is a tuple $\langle \texttt{Act}, \texttt{Subs}, \texttt{Pubs}, \texttt{Seen} \rangle$, where `Act` is a boolean signifying whether the peer is active in the network ($\underline{true}$) or has departed ($\underline{false}$), $\texttt{Subs} \in 2^{\mathcal{T}}$ is a set of topics the peer subscribes to, $\texttt{Pubs} \in 2^{\mathcal{T}}$ is a set of topics the peer publishes in, and `Seen` is a set of full message payloads that have been received. Once a peer joins a `Floodsub` network, its `Pubs` set does not change until it rejoins.

We define the LTS $\mathcal{M}_{\texttt{Broadcastsub}}$ for a `Broadcastsub` as the tuple $\langle S_{\texttt{Broadcastsub}}, \rightarrow_{\texttt{Broadcastsub}}, L_{\texttt{Broadcastsub}} \rangle$ where $S_{\texttt{Broadcastsub}}$ is the broadcast network state which is a map from peers to their states: $\langle \lambda i :: \langle \texttt{Act}, \texttt{Subs}, \texttt{Pubs}, \texttt{Seen} \rangle \rangle$, and $\rightarrow_{\texttt{Broadcastsub}}$ is as defined below:

Rule	Definition
$\texttt{Skip}(s)$	$\underline{\text{skip}}(s)$
$\texttt{Broadcast}(s, m)$	$activep(m.\texttt{Origin}, s) \ \wedge \ m.\texttt{Topic} \in s(m.\texttt{Origin}).\texttt{Pubs} \ \rightarrow$ $\langle \forall k \in \texttt{dom}(s) : activep(k, s) \wedge$ $(m.\texttt{Topic} \in s(k).\texttt{Subs} \vee m.\texttt{Origin} = k) : s(k).\texttt{Seen} :\cup\!= \{m\} \rangle$
$\texttt{BJoin}(s, i, subs, pubs)$	$\neg activep(i, s) \rightarrow s(i) := \langle \underline{true}, subs, pubs, \varnothing \rangle$
$\texttt{BLeave}(s, i)$	$activep(i, s) \rightarrow s(i) := \langle \underline{false}, \varnothing, \varnothing, \varnothing \rangle$
$\texttt{BSubscribe}(s, i, T)$	$activep(i, s) \rightarrow s(i).\texttt{Subs} :\cup\!= T$
$\texttt{BUnsubscribe}(s, i, T)$	$activep(i, s) \rightarrow s(i).\texttt{Subs} :\backslash\!= T$

Table 3.1: $\rightarrow_{\texttt{Broadcastsub}}$ definition.

Suppose B is a single peer in a `Broadcastsub` network. In each of the following examples, unless otherwise stated, we will assume that the message $m = \langle p, t, 1 \rangle$ is carrying a payload p in topic t originating at peer 1. We will also assume that each of the peers subscribe to the topic t. Figure 3.1.1 shows an example of a message broadcast in a `Broadcastsub` consisting of two nodes, 1 and 2.

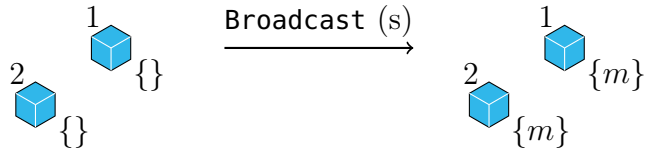


Figure 3.1.1: Example of a message broadcast in a `Broadcastsub` s. A blue hexagon represents a node in a `Broadcastsub`. Its name appears at the top-left, and the set of seen messages at this node appears at the bottom right.

Each of the network protocol abstractions we discuss in this dissertation implement interleaved semantics, *i.e.*, at each step, only one peer can execute an atomic action. `Broadcast` broadcasts a message to all active subscribers in a single step. `BJoin` initialises a peer with $\texttt{Act} := \underline{true}$ and an empty `Seen` set. Its guard $\neg activep(i, s)$ permits both a new peer joining for the first time and a previously departed peer rejoining (since `BLeave` sets $\texttt{Act} := \underline{false}$ but keeps the peer in $\texttt{dom}(s)$). In either case the peer starts with an empty `Seen` set and has no memory of messages seen before departure. `BLeave` sets a peer's `Act` field to $\underline{false}$, and resets `Pubs`, `Subs` and `Seen` sets, marking it as departed. Given a set of topics T, an active peer i either subscribes to or unsubscribes from each of the T topics using the `BSubscribe` and `BUnsubscribe` transitions respectively. The labeling function $L_{\texttt{Broadcastsub}}$ is the `identity` function $\langle \lambda s :: s \rangle$. A starting `Broadcastsub` state is an empty map.

We will now prove that the Commitment⇒Delivery property holds for $\mathcal{M}_{\texttt{Broadcastsub}}$.

Theorem 3.1.1 (Broadcastsub Commitment Implies Delivery)**.** *The Commitment⇒Delivery Property 2.7.14 holds for* $\mathcal{M}_{\texttt{Broadcastsub}}$.

Proof. Let $o = m.\texttt{Origin}$, $c(t) = m \in \sigma(t)(o).\texttt{Seen}$, $e(t) = \mathsf{eligible}(i, m, \sigma(t))$ and $d(t) = m \in \sigma(t)(i).\texttt{Seen}$. We must show that for every message m and every peer i,

$$\mathsf{AG}\Big(\big(\neg c\ \mathsf{U}\ (c \wedge e)\big) \implies \Big(\neg c\ \mathsf{U}\ \Big(c \wedge \big(\mathsf{F}(d) \vee \mathsf{F}(\neg\ e)\big)\Big)\Big)\Big)$$

holds in $\mathcal{M}_{\texttt{Broadcastsub}}$.

Fix any computation path and any state s on that path where the antecedent holds: m was absent from $s(o).\texttt{Seen}$ at all prior states, is now present, and $\mathsf{eligible}(i, m, s)$ holds. By the uniqueness axiom, m is produced at most once. By *Def*. `Broadcast`, when it fires, every active peer k with $\mathit{activep}(k, s)$ and $m.\texttt{Topic} \in s(k).\texttt{Subs}$ or $k = o$ receives m simultaneously.

For the consequent, we must show $\neg(m \in s(o).\texttt{Seen})\ \mathsf{U}\ (m \in s(o).\texttt{Seen} \wedge (\mathsf{F}(m \in s(i).\texttt{Seen}) \vee \mathsf{F}(\neg\, \mathsf{eligible}(i, m))))$. The until is satisfied at the commitment state s itself: $\neg(m \in s(o).\texttt{Seen})$ held at all prior states by the antecedent, and at s we have $m \in s(o).\texttt{Seen}$. Since $\mathsf{eligible}(i, m, s)$ holds, peer i is active and subscribed to $m.\texttt{Topic}$, so $m \in s(i).\texttt{Seen}$ holds at s itself, giving $\mathsf{F}(m \in s(i).\texttt{Seen})$ at step zero.

For states s where the antecedent does not hold, the formula holds vacuously. Since the argument holds at every state on every path, the AG quantifier is satisfied. □

3.2 Formal Description of the Floodsub model

`Floodsub` is our most abstract specification that is also implemented in libp2p. It considers the network topology and message routing which are necessary to broadcast messages in a physical computer network. The libp2p implementation is simple: for each incoming message, a peer forwards it to all of its known peers in the topic of the message. Each peer maintains a cache of previously received messages such that they are not forwarded any further. Also, peers never forward a message back to its source or to the peer they received it from. Peers forwarding messages to all of their outgoing edges leads to flooding, hence the name `Floodsub`.

Since we assume flawless network connectivity, with no disruptions, we allow each message to be forwarded only once to each of the known peers, which we refer to as *neighbors*. We maintain an ordered (by time of arrival) cache of received messages called a *pending* queue, where each of the received messages are stored until forwarded. When a message is forwarded, it is removed from the pending queue and added to the *seen* set. Each of the above mentioned message stores as well as neighbors and their topic subscriptions are stored in the peer's state.

A `Floodsub` state is a partial map from peers to their states. Each peer is identified by a distinct natural number. If a peer joins, it gets assigned a new number, otherwise if it rejoins, it is marked *active*. The state for peer $i \in \mathsf{dom}(s)$ is a tuple $\langle \texttt{Act}, \texttt{Subs}, \texttt{Pubs}, \texttt{Nbrs}, \texttt{Pdg}, \texttt{Seen} \rangle$, where `Act` is a boolean signifying whether the node is active in the network (*true*), or has departed (*false*). `Pubs` is the set of topics the peer publishes in; this is set at join time and never changes thereafter — no transition in `Floodsub`

modifies `Pubs` after a peer has joined. `Subs` is the set of topics to which the peer subscribes and `Nbrs` is a total map of type $\mathcal{T} \to 2^{\mathbb{N}}$ that stores *known neighboring peers* subscribed to that topic; this is not the complete set of all peers subscribed to the topic in the network, but only those directly connected to peer i. Although `Nbrs` is a total map defined per peer, the `Floodsub` protocol preserves the invariant that the neighbor relation is symmetric. `Pdg` is an unbounded First In First Out (FIFO) queue of message payloads that have yet to be forwarded and `Seen` is a set of message payloads that have been received and forwarded.

We define the following atomic predicates:

$$hasMsgp(i, m, s) = m \in s(i).\texttt{Pdg} \cup s(i).\texttt{Seen}$$

$$mNbrp(i, j, m, s) = i \in s(j).\texttt{Nbrs}(m.\texttt{Topic})$$

$$mSubsp(i, m, s) = m.\texttt{Topic} \in s(i).\texttt{Subs}$$

Invariant 3.2.1 (Neighbor symmetry)**.** *The `Floodsub` protocol preserves the following symmetry invariant on the neighbor relation: for all active peers i, j and message m,*

$$mNbrp(i, j, m, s) \iff mNbrp(j, i, m, s).$$

That is, if peer j is a neighbor of peer i on $m.$`Topic`, then peer i is a neighbor of peer j on $m.$`Topic`. This is maintained by `FJoin` and `FSubscribe`, which register i in the neighbor sets of all peers in `Nbrs`(t), and by `FLeave` and `FUnsubscribe`, which remove i from those same sets.

Given any `Floodsub` state s, let

Definition 3.2.15 (Pending messages)
$$\mathcal{P}(s) = \bigcup_{j \,:\, activep(j,s)} s(j).\texttt{Pdg}$$

We define the LTS for a `Floodsub` as the tuple $\mathcal{M}_{\texttt{Floodsub}} = \langle S_{\texttt{Floodsub}}, \to_{\texttt{Floodsub}}, L_{\texttt{Floodsub}} \rangle$ where $S_{\texttt{Floodsub}}$ is the `Floodsub` network state which is a function $\langle \lambda i :: \langle \texttt{Act}, \texttt{Subs}, \texttt{Pubs}, \texttt{Nbrs}, \texttt{Pdg}, \texttt{Seen} \rangle \rangle$, $\to_{\texttt{Floodsub}}$ is as defined in Table 3.2.

`FProduce` produces a new message at peer i; by the uniqueness axiom, the message is globally fresh and not present anywhere in the network. `FForward` forwards the message from the front of the non-empty `Pdg` queue to all the neighboring subscribers, except those who have already seen it. Notice that if m is in `Pdg`, then it is not in `Seen`, and vice versa, because we model peers as discarding any message they receive if they already have it in their state. `FJoin` allows a peer to join (or rejoin) the network with a given set of subscriptions, publications, and neighbors. Note that `Floodsub` maintains the invariant that a peer does not appear in its own `Nbrs` map. In practice, a *peer discovery* mechanism — such as a distributed hash table or a rendezvous server — identifies suitable neighbors before the join is initiated. We do not model peer discovery explicitly; instead, we treat the existence of such neighbors as given when a peer joins. `FLeave` allows a peer to leave the network, if it is active. It sets the leaving peer as inactive, and removes it from its neighbor's states. `FReset` resets the state of a peer, and simulates a hard exit, where all state information are lost.

Rule	Definition
$\texttt{FProduce}(s, m)$	$activep(m.\texttt{Origin}, s) \ \wedge \ m.\texttt{Topic} \in s(m.\texttt{Origin}).\texttt{Pubs} \ \rightarrow$ $s(m.\texttt{Origin}).\texttt{Pdg}.\underline{\text{enqueue}}(m)$
$\texttt{FForward}(s, i)$	$activep(i, s) \ \wedge \ \neg s(i).\texttt{Pdg}.\underline{\text{empty}} \rightarrow \underline{\text{let}} \ m = s(i).\texttt{Pdg}.\underline{\text{front}} \ \underline{\text{in}}$ $s(i).\texttt{Pdg}.\underline{\text{dequeue}} \ ⨟ \ s(i).\texttt{Seen} :\cup\!= \{m\} \ ⨟$ $\langle \forall k \ : \ mNbrp(k, i, m, s) \ \wedge \ \neg hasMsgp(k, m, s) \ :$ $s(k).\texttt{Pdg}.\underline{\text{enqueue}}(m) \rangle$
$\texttt{FJoin}(s, i, subs, pubs, nbrs)$	$\neg activep(i, s) \ \wedge \ i \notin \texttt{rng}(nbrs) \rightarrow$ $\langle \forall t, k \ : \ (t \in subs \cup pubs) \wedge (k \in nbrs(t)) \ : \ (s(k).\texttt{Nbrs}(t) :\cup\!= \{i\}) \rangle \ ⨟$ $s(i) := \langle \underline{true}, subs, pubs, nbrs, [\,], \varnothing \rangle$
$\texttt{FLeave}(s, i)$	$activep(i, s) \rightarrow$ $\langle \forall t, k \ : \ t \in s(i).\texttt{Subs} \wedge \ k \in s(i).\texttt{Nbrs}(t) \ : \ s(k).\texttt{Nbrs}(t) :\backslash\!= \{i\} \rangle ⨟$ $s(i).\texttt{Act} := \underline{false}$
$\texttt{FReset}(s, i)$	$s(i) := \langle \underline{false}, \varnothing, \varnothing, \varnothing, [\,], \varnothing \rangle$
$\texttt{FSubscribe}(s, i, T)$	$activep(i, s) \rightarrow s(i).\texttt{Subs} :\cup\!= T \ ⨟$ $\langle \forall t, k \ : \ t \in T \wedge \ k \in \texttt{rng}(s(i).\texttt{Nbrs}) \ : \ s(k).\texttt{Nbrs}(t) :\cup\!= \{i\} \rangle$
$\texttt{FUnsubscribe}(s, i, T)$	$activep(i, s) \rightarrow s(i).\texttt{Subs} :\backslash\!= T \ ⨟$ $\langle \forall t, k \ : \ t \in T \wedge \ k \in \texttt{rng}(s(i).\texttt{Nbrs}) \ : \ s(k).\texttt{Nbrs}(t) :\backslash\!= \{i\} \rangle$

Table 3.2: $\rightarrow_{\texttt{Floodsub}}$ definition.

`FUnsubscribe` allows a peer to unsubscribe from a set of topics. `FSubscribe` allows a peer to subscribe to a given set of topics. We assume that the start state of a `Floodsub` network is an empty map.

Definition 3.2.16 (Labeling function for `Floodsub`)
The labeling function $L_{\texttt{Floodsub}}$ is defined as:

$$L_{\texttt{Floodsub}}(s) = \langle \lambda i \in \mathsf{dom}(s) :: \langle s(i).\texttt{Act}, s(i).\texttt{Subs}, s(i).\texttt{Pubs}, s(i).\texttt{Nbrs}, s(i).\texttt{Seen} \rangle \rangle$$

That is, $L_{\texttt{Floodsub}}$ projects out the `Pdg` queue, retaining only the components relevant to observable network state.

3.3 Correctness Conditions for Floodsub

`Broadcastsub` is a specification. In a more realistic network like `Floodsub`, where peers need to communicate on a physical network, a message is not received by its subscribers instantaneously. It might need to hop from the originating peer to its neighboring peers and so on, until it reaches its subscribers.

In the text below, we will use the following pictorial notation to represent the state of `Floodsub` nodes.

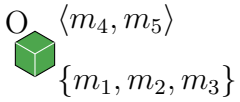


Figure 3.3.1: The green hexagon represents a single `Floodsub` node, with its name O displayed at the top. The contents of its `Pdg` and `Seen` seen sets are shown at the top right and bottom right, respectively.

Suppose O, X and Y are peers in a `Broadcastsub` network which subscribe to the same topics. In each of the following examples, unless otherwise stated, we will assume that the message $m = \langle p, t, O \rangle$ is carrying a payload p in topic t originating at peer O. We will also assume that each of the peers subscribe to the topic t. Figure 3.3.2 shows an example of a message broadcast in a `Broadcastsub`.

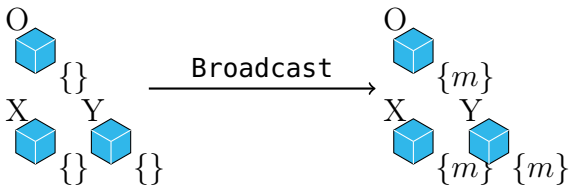


Figure 3.3.2: Example of a message broadcast in a `Broadcastsub`

We will go through a series of examples to understand the conditions under which we can show correctness.

Example 1. Consider two peers O and X as shown. Suppose that O and X subscribe to $m.\texttt{Topic}$ but are not connected in topic $m.\texttt{Topic}$ *i.e.* $O.\texttt{Nbrs}(m.\texttt{Topic}) = \varnothing = X.\texttt{Nbrs}(t)$.

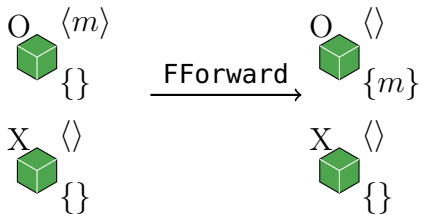


Figure 3.3.3: Example showing message dissemination failure in a disconnected `Floodsub`

In this example, we see that m was not fully propagated, with X yet to receive it. Hence it is clear that the subscribers of each topic should be connected in that topic.

Example 2. Suppose we have a `Floodsub` consisting of peers O, X and Y as shown, all of which are connected. However, before X could forward the message m to its neighboring peer Y, it left the network. As a result m was not fully propagated.

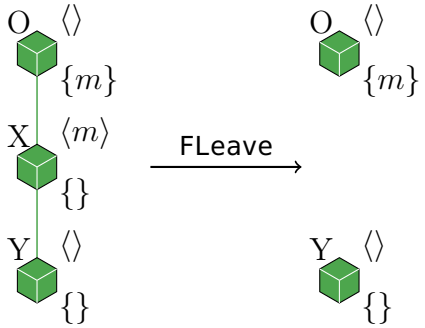


Figure 3.3.4: Example showing message dissemination failure due to a leaving peer

The issue here is that even though we started with a connected network, `Floodsub` peers are free to leave or join the network or subscribe to or unsubscribe from topics at any time, affecting network connectivity. There are four transitions that can potentially change the path of message flow: `FLeave`, `FJoin`, `FSubscribe` and `FUnsubscribe`. We need to make sure that the network remains connected in each topic, while allowing all of these transitions. This does not mean that we require that the network be connected in every topic at all times. What we need is for the network to be dynamically connected over time, in each topic *i.e.*, have temporal paths as defined below.

Definition 3.3.17 (Temporal Graph)

A *Temporal Graph* is an infinite sequence of graphs $[G_0, G_1, \ldots]$ where each graph $G_t = \langle V_t, E_t \rangle$ is a pair of sets of its vertices V_t and edges E_t at step $t \in \mathbb{N}$.

Below we define the notion of a temporal path.

Definition 3.3.18 (Temporal Path)

Given a temporal graph $G = [\langle V_0, E_0\rangle, \langle V_1, E_1\rangle, \ldots]$, a *temporal path* from node u to node v in G is a finite sequence of edges $P = [\langle v_0, v_1\rangle, \langle v_1, v_2\rangle, \ldots, \langle v_{n-1}, v_n\rangle]$ such that

- $u = v_0 \ \wedge \ v = v_n$, and
- $\langle \forall \langle v_i, v_{i+1}\rangle \in P :: \langle v_i, v_{i+1}\rangle \in E_i\rangle$.

We write $u \xrightarrow{G}{}^{*} v$ to denote that a temporal path from u to v exists in G.

Definition 3.3.19 (Temporal Graph for message m)

Given an infinite path $s_0 \to s_1 \to s_2 \to \cdots$ of `Floodsub` transitions and a message m, let $t_0 < t_1 < t_2 < \cdots$ be the subsequence of indices at which an `FForward` transition that forwards m occurs, *i.e.* $s_{t_j} \xrightarrow{\texttt{FForward}} s_{t_j+1}$ for each $j \in \mathbb{N}$. If no such indices exist (because m was never produced, or was produced but never forwarded), define ${}^{\mathit{Flood}}G^m = [\langle \varnothing, \varnothing\rangle, \langle \varnothing, \varnothing\rangle, \ldots]$, the infinite sequence of empty graphs. Otherwise, the temporal graph for m is the infinite sequence of graphs ${}^{\mathit{Flood}}G^m = [\langle V_0, E_0\rangle, \langle V_1, E_1\rangle, \ldots]$ where, for each $j \in \mathbb{N}$:

- $V_j = \{i \mid \mathsf{eligible}(i, m, s_{t_j}) \vee (m = s_{t_j}(i).\mathsf{Pdg}.\underline{\text{front}} \wedge \mathit{activep}(i, s_{t_j}))\}$,
- $E_j = \{\langle k, i\rangle \mid \ \mathit{activep}(k, s_{t_j}) \ \wedge \ \neg s_{t_j}(k).\mathsf{Pdg}.\underline{\text{empty}} \wedge \ m = s_{t_j}(k).\mathsf{Pdg}.\underline{\text{front}} \wedge s_{t_j+1} = \texttt{FForward}(s_{t_j}, k) \wedge \ \mathit{mNbrp}(k, i, m, s_{t_j}) \ \wedge \ \neg \mathit{hasMsgp}(i, m, s_{t_j})\}$

If only finitely many such indices exist, say $n > 0$, the sequence is extended by setting $V_j = V_{n-1}$ and $E_j = \emptyset$ for all $j \geq n$. Thus E_j records the set of edges $\langle k, i\rangle$ along which m was actually forwarded at the j-th `FForward` step for m, and $E_j = \emptyset$ at extended steps where m has stopped propagating.

We define the temporal connectedness condition *TConnectedp* as a condition on fullpaths of the `Floodsub` transition system.

Definition 3.3.20 (Temporal Connectedness Condition)

A fullpath σ of `Floodsub` satisfies $TConnectedp(\sigma)$ iff for every message m, originating peer $o = m.\texttt{Origin}$, and peer i, letting $c(t) = m \in \sigma(t)(o).\texttt{Seen}$ and $e(t) = \mathsf{eligible}(i, m, \sigma(t))$:

$$\Big(\big(\neg c \;\mathsf{U}\; (c \wedge e)\big) \implies \Big(\neg c \;\mathsf{U}\; \Big(c \wedge \big(o \xrightarrow{{}^{Flood}G_\sigma^m}{}^{*}\; i \vee \mathsf{F}(\neg\, e)\big)\Big)\Big)\Big)$$

That is, if there exists a unique time when m is forwarded by o (and therefore added to $o.\texttt{Seen}$), and i is eligible to receive it, then either there exists a temporal path from o to i tracing the sequence of actual `FForward` hops by which m propagates to i, or i eventually becomes ineligible (by departing or unsubscribing), in which case no delivery obligation applies.
A `Floodsub` transition system satisfies $TConnectedp$ iff every fullpath σ satisfies $TConnectedp(\sigma)$.

For any message m and eligible peer i, the condition says: once m is committed by its originator o (i.e., $m \in o.\texttt{Seen}$), either there exists a temporal path in ${}^{Flood}G_\sigma^m$ from o to i

Crucially, the path need not exist at any instant in the network topology. It is enough that each hop occurs at a later step than the previous one: peer o forwards m to k_1 at step t_1, then k_1 forwards to k_2 at a later step $t_2 > t_1$, so long as m eventually reaches i. Temporal paths for messages from their originating peers to eligible peers are unique along fullpaths because each peer receives m at most once.

3.4 Choosing a refinement map

A refinement proof is relative to a refinement map that maps the implementation (`Floodsub`) states to the specification (`Broadcastsub`) states. It shows us how to view a `Broadcastsub` state as a `Floodsub` state *e.g.* the refinement map needs to hide certain components of the `Floodsub` peer state such as `Nbrs` and `Pdg` sets which do not appear in a `Broadcastsub` state. Simple refinement maps with a clear relationship between implementation states and their image under the map are best [55]. We consider two refinement maps : (1) commitment refinement map and (2) flushing refinement map, both of which have been used in the verification of pipelined microprocessors [29, 54, 56–58].

We define our commitment refinement map `f2b` over all peers in a `Floodsub` state as follows:

Definition 3.4.21 (Refinement map from `Floodsub` to `Broadcastsub` states)

$$\texttt{f2b}(s) = \Big\langle \lambda i \in \mathsf{dom}(s) :: \begin{cases} \langle \underline{\mathit{true}}, \\ \quad s(i).\texttt{Subs} \cup \\ \quad\quad \underbrace{\{\, m.\texttt{Topic} \mid \mathit{hasMsgp}(i, m, s) \wedge m \in \mathcal{P}(s) \wedge m.\texttt{Origin} \neq i \,\}}_{\text{subscriptions}}, \\ \quad s(i).\texttt{Pubs}, \\ \quad s(i).\texttt{Seen} \setminus \mathcal{P}(s) \rangle & \underline{\text{if}}\ s(i).\texttt{Act} \vee i \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(s)\} \\ \\ \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing \rangle\ \underline{\text{otherwise}} \end{cases} \Big\rangle$$

Since $\mathcal{P}(s)$ is defined over active peers only, an inactive peer i appears in $\mathsf{dom}(\texttt{f2b}(s))$ only if some *active* peer holds a pending message originated at i.

In a `Floodsub` state, message dissemination is not an atomic operation, unlike `Broadcast` in a `Broadcastsub` state. Instead, it unfolds over multiple transitions, during which peers that forwarded such messages may unsubscribe, or peers that produced the messages may leave the network, even though their fingerprints (*i.e.*, pending messages) remain in the network. Hence, we define our refinement map in such a way that while a message is pending and does not show in the image of the refinement map, its originating node does. Similarly, topic subscriptions of peers on the temporal path of pending messages are preserved by `f2b` until such messages are disseminated. Because a message originator receives its own message by virtue of publishing, not subscribing, we do not need to preserve its topic subscription in `f2b` — the originator's `Pubs` set, which is invariant, already accounts for its membership. It is clear that given a `Floodsub` state s, there are no pending messages in the `Seen` sets of `Broadcastsub` peer states of $\texttt{f2b}(s)$. The only messages present are those that have been disseminated or committed. Therefore, this is an example of the commitment approach [55].

Another approach is the flushing refinement map which maps a given `Floodsub` state to a `Broadcastsub` state after all the `Pdg` queues have been flushed. We will show why we did not choose this refinement map with an example. Suppose a message m is pending in a `Floodsub` state s. Let there be no path for m to reach all of its subscribers in s, unless another node p joins in a future state and provides a temporal path for m. If we use the flushing refinement approach, the propagation of m stalls and p will never receive m in the future. This example brings up several issues: (1) **Expressiveness**: the mapping is too coarse, matching only quiet `Floodsub` states where there are no in-flight messages, (2) **Safety reasoning**: since the mapping is too coarse, it might miss safety property violations in the intermediate states, and (3) **Liveness reasoning**: we are concerned about liveness properties like eventual delivery of messages to subscribers, which is ignored.

3.4.1 Choosing a Notion of Correctness

We want to establish that `Floodsub` correctly implements `Broadcastsub` in the strongest sense possible. We use *skipping simulation refinement* [26–28], which allows the concrete system to take several steps for every one abstract step, and permits the abstract system

to skip over segments where no corresponding concrete step exists. This is strictly stronger than trace inclusion and subsumes stuttering simulation.

Why not bisimulation or stuttering simulation? Both are ruled out by the same asymmetry. In `Floodsub`, a peer i may leave the network while messages it originated are still pending in other peers' `Pdg` queues. By definition of `f2b` , i is active in $\mathsf{dom}(\texttt{f2b}(s))$ for as long as such messages exist, because the domain condition $s(i).\texttt{Act} \vee i \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(s)\}$ retains departed originators of pending messages. The resulting abstract `Broadcastsub` state therefore contains a peer with $s(i).\texttt{Act} = \underline{\mathit{false}}$. But `Broadcastsub`'s `Broadcast` transition requires $\mathit{activep}(m.\texttt{Origin}, \texttt{f2b}(s))$: when `Floodsub` commits a message whose originator has already departed, no matching abstract step can fire. This means the concrete system must advance past the point where the abstract step would occur before any abstract transition is enabled. Neither bisimulation nor stuttering simulation can accommodate this: both require the abstract system to keep pace with the concrete one, either step for step or with a decreasing rank. Skipping refinement resolves this by allowing the concrete path to run ahead, with the simulation relation only required to hold at matching endpoints.

Hence we show skipping refinement, formalised as a Reduced Well-Founded Skipping Refinement (RWFSK) 2.3.8.

3.5 The Refinement Proof

We prove that `Floodsub` is a skipping refinement of `Broadcastsub`. `f2b` is our refinement map from good `Floodsub` states to `Broadcastsub` states. To show that `Floodsub` is a skipping refinement of `Broadcastsub`, we need to define a relation B such that $\langle \forall s \in S_{\texttt{Floodsub}} :: sB\texttt{f2b}(s)\rangle$ and show that B is a Reduced Well Formed Skipping simulation (RWFSK) on the LTS $\langle S, \rightarrow, \mathcal{L}\rangle$ where $S = S_{\texttt{Floodsub}} \uplus S_{\texttt{Broadcastsub}}$, $\rightarrow = \rightarrow_{\texttt{Floodsub}} \uplus \rightarrow_{\texttt{Broadcastsub}}$ and $\mathcal{L}(s) = L_{\texttt{Broadcastsub}}(\texttt{f2b}(s))$ where $s \in S_{\texttt{Floodsub}}$ and $\mathcal{L}(s) = L_{\texttt{Broadcastsub}}(s)$ otherwise. Given $s, w \in S$, and recognizer functions fp and bp for `Floodsub` and `Broadcastsub` states respectively, we define B as follows.

Definition 3.5.22 (The Relation B)

$$sBw \iff \begin{cases} & s = w \\ \vee & \mathsf{fp}(s) \wedge\ \mathsf{bp}(w) \wedge\ \texttt{f2b}(s) = w \end{cases}$$

Theorem 3.5.1. *If* $\mathcal{M}_{\texttt{Floodsub}}$ *satisfies TConnectedp then* $\mathcal{M}_{\texttt{Floodsub}} \lesssim_{\texttt{f2b}} \mathcal{M}_{\texttt{Broadcastsub}}$.

Proof. $\langle \forall s \in S_{\texttt{Floodsub}} :: sB(\texttt{f2b}(s))\rangle$ follows from the definition of B.

To prove that B is a Well founded skipping relation on the LTS $\langle S, \rightarrow, \mathcal{L}\rangle$, we show the following.

Goal 3.5.1.1. ***RWFSK1.*** *(Definition 2.3.8)*

Proof. Given sBw, we have

- if $s = w$ then $\mathcal{L}(s) = \mathcal{L}(w)$
- if $\mathsf{fp}(s) \wedge \mathsf{bp}(w)$ then $\mathcal{L}(s) = L_{\texttt{Broadcastsub}}(\texttt{f2b}(s)) = L_{\texttt{Broadcastsub}}(w) = \mathcal{L}(w)$

□

We will need the following lemmas to prove RWFSK2.

Lemma 3.5.1.1.

$$\langle \forall s, i \ :: \mathit{fp}(s) \wedge \ \mathit{activep}(i, s) \wedge \ \neg s(i).\mathit{Pdg}.\underline{\mathit{empty}} \Rightarrow \\ \mathcal{P}(\texttt{FForward}(s, i)) \neq \mathcal{P}(s) \ \Leftrightarrow \mathcal{P}(\texttt{FForward}(s, i)) = \mathcal{P}(s) \setminus \{s(i).\mathit{Pdg}.\underline{\mathit{front}}\}\rangle$$

Proof. Let $m = s(i).\texttt{Pdg}.\underline{\text{front}}$ and $u = \texttt{FForward}(s, i)$.

Pong ($\Leftarrow$): If $\mathcal{P}(u) = \mathcal{P}(s) \setminus \{m\}$ then clearly $\mathcal{P}(u) \neq \mathcal{P}(s)$ since $m \in \mathcal{P}(s)$.

Ping ($\Rightarrow$): Suppose $\mathcal{P}(u) \neq \mathcal{P}(s)$. By the `FForward` rule (Table 3.2), executing `FForward` at peer i dequeues m from i's `Pdg` and delivers it to each neighbor $j \in s(i).\texttt{Nbrs}(m.\texttt{Topic})$ that does not already have m. Any such neighbor j gains m in its `Pdg`, so $m \in \mathcal{P}(u)$ if any neighbor received it. Since $\mathcal{P}(u) \neq \mathcal{P}(s)$, no neighbor received m, meaning every neighbor already held m in its `Pdg` or `Seen` set before the transition. Furthermore, $i \notin s(i).\texttt{Nbrs}(m.\texttt{Topic})$ by invariant, so i does not re-receive its own forwarded message. Therefore m is removed from i's `Pdg` and added to no other peer's `Pdg`, so m leaves $\mathcal{P}$ entirely: $\mathcal{P}(u) = \mathcal{P}(s) \setminus \{m\}$. □

Lemma 3.5.1.2. *If* $\mathit{fp}(s) \wedge \ \mathit{fp}(u) \wedge \ \mathit{dom}(u) = \mathit{dom}(s)$, *then* $\mathit{dom}(\mathit{f2b}(u)) = \mathit{dom}(\mathit{f2b}(s))$.

Proof.

$\mathsf{dom}(\texttt{f2b}(u))$

$\equiv$ *Def.* `f2b`

$\mathsf{dom}(u)$

$\equiv \mathsf{dom}(u) = \mathsf{dom}(s)$

$\mathsf{dom}(s)$

$\equiv$ *Def.* `f2b`

$\mathsf{dom}(\texttt{f2b}(s))$

□

Goal 3.5.1.2. ***RWFSK2.*** *(Definition 2.3.8)*

Proof. Given states $s, w, u \in S \ \wedge \ sBw \ \wedge \ s \to u$. We consider cases on the B relation.

Case 1. If $s = w$ then for any transition $s \to u$, we have $w \to v$ where $u = v$ and hence uBv.

Case 2. Otherwise, let $\mathsf{fp}(s) \wedge \mathsf{bp}(w) \wedge \ \mathtt{f2b}(s) = w$. Since $\mathsf{fp}(s)$, let $s \to_{\mathtt{Floodsub}} u$.
We consider cases on $\to_{\mathtt{Floodsub}}$.

- Let $activep(m.\mathtt{Origin}, s) \ \wedge \ m.\mathtt{Topic} \in s(m.\mathtt{Origin}).\mathtt{Pubs} \ \wedge \ u = \mathtt{FProduce}(s, m)$.
  By the uniqueness axiom, $m \notin \bigcup_{i \in \mathsf{dom}(s)} (s(i).\mathsf{Pdg} \cup s(i).\mathtt{Seen})$, so $\mathcal{P}(u) = \mathcal{P}(s) \cup \{m\}$ and $m \notin \mathcal{P}(s)$.

  $\mathtt{f2b}(u)$

  $\equiv$ *Def*. $\mathtt{f2b}$

  $\langle \lambda i \in \mathsf{dom}(\mathtt{f2b}(u)) :: \underline{\mathit{true}},$
  $u(i).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(i, e, u) \wedge e \in \mathcal{P}(u) \wedge e.\mathtt{Origin} \neq i\},$
  $u(i).\mathtt{Pubs},$
  $u(i).\mathtt{Seen} \setminus \mathcal{P}(u)\rangle$ $\underline{\text{if}}$ $u(i).\mathtt{Act} \vee i \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(u)\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

  $\equiv \mathsf{dom}(u) = \mathsf{dom}(s)$, Lemma 3.5.1.2, $\mathsf{dom}(\mathtt{f2b}(u)) = \mathsf{dom}(\mathtt{f2b}(s))$

  $\langle \lambda i \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{\mathit{true}},$
  $u(i).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(i, e, u) \wedge e \in \mathcal{P}(u) \wedge e.\mathtt{Origin} \neq i\},$
  $u(i).\mathtt{Pubs},$
  $u(i).\mathtt{Seen} \setminus \mathcal{P}(u)\rangle$ $\underline{\text{if}}$ $u(i).\mathtt{Act} \vee i \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(u)\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

  $\equiv$ *Def*. $\mathtt{FProduce}$, $\mathcal{P}(u) = \mathcal{P}(s) \cup \{m\}, activep(i, s)$

  $\langle \lambda i \in \mathsf{dom}(\mathtt{f2b}(s)) :: \underline{\mathit{true}},$
  $s(i).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(i, e, s) \wedge e \in \mathcal{P}(s) \cup \{m\} \wedge e.\mathtt{Origin} \neq i\},$
  $s(i).\mathtt{Pubs},$
  $s(i).\mathtt{Seen} \setminus (\mathcal{P}(s) \cup \{m\})\rangle$ $\underline{\text{if}}$ $s(i).\mathtt{Act} \vee i \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(s) \cup \{m\}\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

  $\equiv m$ uniqueness axiom

  $\langle \lambda i \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{\mathit{true}},$
  $s(i).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(i, e, s) \wedge e \in \mathcal{P}(s) \cup \{m\} \wedge e.\mathtt{Origin} \neq i\},$
  $s(i).\mathtt{Pubs},$
  $s(i).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $s(i).\mathtt{Act} \vee i \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(s) \cup \{m\}\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

$\equiv$ only i has m , $activep(i, s)$

$\langle \lambda i \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{true},$
$s(i).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(i, e, s) \wedge e \in \mathcal{P}(s) \wedge e.\mathtt{Origin} \neq i\},$
$s(i).\mathtt{Pubs},$
$s(i).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $s(i).\mathtt{Act} \vee i \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(s)\}$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

$\equiv$ *Def*. $\mathtt{f2b}$, $\quad w = \mathtt{f2b}(s)$

w

Let $v = \mathtt{Skip}(w) = w = \mathtt{f2b}(u)$. Therefore uBv.

- Let $activep(i, s) \ \wedge \ \neg s(i).\mathtt{Pdg}.\underline{\text{empty}} \ \wedge \ u = \mathtt{FForward}(s, i)$.
  Let $m = s(i).\mathtt{Pdg}.\underline{\text{front}}$.

  Case (i): Let $\mathcal{P}(u) = \mathcal{P}(s)$.

  $\mathtt{f2b}(u)$

  $\equiv$ *Def*. $\mathtt{f2b}$

  $\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(u)) :: \langle \underline{true},$
  $u(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(j, e, u) \wedge e \in \mathcal{P}(u) \wedge e.\mathtt{Origin} \neq j\},$
  $u(j).\mathtt{Pubs},$
  $u(j).\mathtt{Seen} \setminus \mathcal{P}(u)\rangle$ $\underline{\text{if}}$ $u(j).\mathtt{Act} \vee j \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(u)\}$
  $\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

  $\equiv \mathsf{dom}(u) = \mathsf{dom}(s)$, Lemma 3.5.1.2, $\mathsf{dom}(\mathtt{f2b}(u)) = \mathsf{dom}(\mathtt{f2b}(s)), \mathcal{P}(u) = \mathcal{P}(s)$

  $\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{true},$
  $u(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(j, e, u) \wedge e \in \mathcal{P}(s) \wedge e.\mathtt{Origin} \neq j\},$
  $u(j).\mathtt{Pubs},$
  $u(j).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $u(j).\mathtt{Act} \vee j \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(s)\}$
  $\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

  $\equiv$ *Def*.$\mathtt{FForward}, hasMsgp(j, e, u) = hasMsgp(j, e, s)$

  $\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{true},$
  $s(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(j, e, s) \wedge e \in \mathcal{P}(s) \wedge e.\mathtt{Origin} \neq j\},$
  $s(j).\mathtt{Pubs},$
  $(\underline{\text{if}}\ j = i\ \underline{\text{then}}\ s(i).\mathtt{Seen} \cup \{m\}\ \underline{\text{else}}\ s(j).\mathtt{Seen}) \setminus \mathcal{P}(s)\rangle$
  $\underline{\text{if}}$ $s(j).\mathtt{Act} \vee j \in \{e.\mathtt{Origin} \mid e \in \mathcal{P}(s)\}$
  $\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

  $\equiv$ *Lemma* 3.5.1.1, $m \in \mathcal{P}(s)$

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \wedge e \in \mathcal{P}(s) \wedge e.\texttt{Origin} \neq j\},$
$s(j).\texttt{Pubs},$
$s(j).\texttt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

$\equiv \mathit{Def}.\texttt{f2b}, \texttt{f2b}(s) = w$

w

Let $v = \texttt{Skip}(w)$. Hence uBv.

Case (ii): Otherwise $\mathcal{P}(u) \neq \mathcal{P}(s)$. Consider the following:

- Consider the case where $\neg s(m.\texttt{Origin}).\texttt{Act}$. Then $\mathit{activep}(m.\texttt{Origin}, \texttt{f2b}(s))$, because `f2b` preserves originating peers of pending messages, however $\neg \mathit{activep}(m.\texttt{Origin}, \texttt{f2b}(u))$, as $m \notin \mathcal{P}(u)$. Hence the single `FForward` step on the `Floodsub` side will need to skip over multiple steps on the `Broadcastsub` side: m getting broadcast and $m.\texttt{Origin}$ becoming inactive.
- Similarly, node subscriptions under the refinement map `f2b` include topic subscriptions for pending messages that are present in the corresponding `Pdg` queue or `Seen` set. Once these messages are no longer pending, and if a `Floodsub` node is already unsubscribed from the topics of such messages, the refinement map `f2b` will ignore these topic subscriptions, due to which there may be nodes that do not subscribe to $m.\texttt{Topic}$ but have m in their `Seen` set. Again, this points to the need for multiple skipped steps on the `Broadcastsub` side where corresponding nodes will have to unsubscribe after receiving m.
- While m is pending, new nodes can join that may either be on m's temporal path or not. There might be some nodes in u which subscribe to $m.\texttt{Topic}$ such that they were not on m's temporal path. Such nodes will not have seen m in state u. However, a `Broadcast` on $\texttt{f2b}(s)$ will ensure that all subscribers of m receive m. Similarly, there might be some nodes that joined while m was propagating through the network, and did receive it due to being temporally connected to m. So subscription to $m.\texttt{Topic}$ alone will not inform which nodes receive m. We will have to check that which nodes actually received m on the `Floodsub` side, and then `Broadcast` to them.

Due to the above considerations, we will need to take some extra steps from w to reach a state v that is related to u, as shown in the following algorithm:

Algorithm 1 BroadcastM

Require: State w, Message m, and Target Configuration u

Ensure: A state v produced after `Broadcastsub` transitions where m is broadcast

Input: w, m, u

$v \leftarrow w$

while $\langle \exists i :: activep(i,u) \ \wedge \ mSubsp(i,m,u) \ \wedge \ m \notin u(i).\texttt{Seen} \rangle$ **do**

$v \leftarrow \texttt{BUnsubscribe}(v, i, \{m.\texttt{Topic}\})$ ▷ $mSubsp(i,m,u) \Rightarrow m.\texttt{Topic} \in v(i).\texttt{Subs}$ from Def.`f2b` . These nodes should not receive m, so unsubscribing.

end while

$v \leftarrow \texttt{Broadcast}(v, m)$

▷ For pending m, `f2b` keeps $m.\texttt{Origin}$ and discards m from `Seen` sets of mapped nodes

while $\langle \exists i :: activep(i,u) \ \wedge \ mSubsp(i,m,u) \ \wedge \ m \notin u(i).\texttt{Seen} \rangle$ **do**

$v \leftarrow \texttt{BSubscribe}(v, i, \{m.\texttt{Topic}\})$ ▷ Revert previous unsubscriptions

end while

while $\langle \exists i :: activep(i,u) \ \wedge \ \neg mSubsp(i,m,u) \ \wedge \ m \in u(i).\texttt{Seen} \rangle$ **do**

$v \leftarrow \texttt{BUnsubscribe}(v, i, \{m.\texttt{Topic}\})$ ▷ $mSubsp(i,m,u) \wedge m \in u(i).\texttt{Seen} \Rightarrow m.\texttt{Topic} \in v(i).\texttt{Subs}$ from Def.`f2b` . These nodes should not be subscribed, as in u.

end while

if $\neg u(m.\texttt{Origin}).\texttt{Act}$ **then**

$v \leftarrow \texttt{BLeave}(v, m.\texttt{Origin})$ ▷ $activep(m.\texttt{Origin}, v)$ from Def. `f2b` .

end if

return v

Each step in Algorithm 1 is enabled by construction from Definition 3.4.21, and the algorithm preserves `bp` invariants. We justify guards for each of the `Broadcastsub` steps we take in the previous Algorithm 1. By inspection, the algorithm is terminating as the total number of nodes in a network is finite and each while-loop processes each node at most once. Let v be the `Broadcastsub` state returned from running Algorithm 1 on w. We will prove uBv.

From Lemma 3.5.1.1, $\mathcal{P}(u) = \mathcal{P}(s) \setminus \{m\}$.

$\texttt{f2b}(u)$

$\equiv$ *Def.* `f2b`

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(u)) :: \langle \underline{true},$

$u(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid hasMsgp(j,e,u) \wedge e \in \mathcal{P}(u) \wedge e.\texttt{Origin} \neq j\},$

$u(j).\texttt{Pubs},$

$u(j).\texttt{Seen} \setminus \mathcal{P}(u) \rangle$ $\underline{\text{if}}$ $u(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(u)\}$

$\langle \underline{false}, \varnothing, \varnothing, \varnothing \rangle$ $\underline{\text{otherwise}} \rangle$

$\equiv \mathsf{dom}(u) = \mathsf{dom}(s)$, Lemma 3.5.1.2, $\mathsf{dom}(\texttt{f2b}(u)) = \mathsf{dom}(\texttt{f2b}(s))$, $\mathcal{P}(u) = \mathcal{P}(s) \setminus \{m\}$

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{true},$

$u(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid hasMsgp(j,e,u) \wedge e \in (\mathcal{P}(s) \setminus \{m\}) \wedge e.\texttt{Origin} \neq j\},$

$u(j).\texttt{Pubs},$
$u(j).\texttt{Seen} \setminus (\mathcal{P}(s) \setminus \{m\})\rangle$ if $u(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(s) \setminus \{m\}\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv$ $\mathit{Def}.\texttt{FForward}, \mathit{hasMsgp}(j,e,u) = \mathit{hasMsgp}(j,e,s)$

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j,e,s) \wedge e \in (\mathcal{P}(s) \setminus \{m\}) \wedge e.\texttt{Origin} \neq j\},$
$s(j).\texttt{Pubs},$
$(\underline{\text{if}}\ j = i\ \underline{\text{then}}\ s(i).\texttt{Seen} \cup \{m\}\ \underline{\text{else}}\ s(j).\texttt{Seen}) \setminus (\mathcal{P}(s) \setminus \{m\})\rangle$
$\underline{\text{if}}\ s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(s) \setminus \{m\}\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv$ Simplify

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j,e,s) \wedge e \in \mathcal{P}(s) \wedge e \neq m \wedge e.\texttt{Origin} \neq j\},$
$s(j).\texttt{Pubs},$
$(\underline{\text{if}}\ (j = i \vee m \in s(j).\texttt{Seen})\ \underline{\text{then}}\ (s(j).\texttt{Seen} \setminus \mathcal{P}(s)) \cup \{m\}$
$\underline{\text{else}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s))\rangle$ if $s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

Let v be the state returned by running Algorithm 1 on state w.

We consider the following exhaustive cases on all the nodes:

– Let $m \in u(j).\texttt{Seen} \wedge m.\texttt{Topic} \in u(j).\texttt{Subs}$: Then $m \in s(j).\texttt{Subs}$ too as FForward does not affect subscriptions.

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(i,e,u) \wedge e \in \mathcal{P}(s) \wedge e.\texttt{Origin} \neq i\},$
$s(j).\texttt{Pubs},$
$(\underline{\text{if}}\ (j = i \vee m \in s(j).\texttt{Seen})\ \underline{\text{then}}\ (s(j).\texttt{Seen} \setminus \mathcal{P}(s)) \cup \{m\}$
$\underline{\text{else}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s))\rangle$ if $s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv$ $m \in u(j).\texttt{Seen} \Rightarrow (j = i \vee m \in s(j).\texttt{Seen})$, Simplify

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(i,e,u) \wedge e \in \mathcal{P}(s) \wedge e.\texttt{Origin} \neq i\},$
$s(j).\texttt{Pubs},$
$(s(j).\texttt{Seen} \setminus \mathcal{P}(s)) \cup \{m\}\ \rangle$ if $s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv$ Def. `f2b`, Algorithm 1, Def. `Broadcast`

v

For the nodes under consideration, Algorithm 1 is the `Broadcast` step. If $\neg u(m.\texttt{Origin}).\texttt{Act}$, a `BLeave` step follows.

– Let $m \in u(j).\texttt{Seen} \wedge m.\texttt{Topic} \notin u(j).\texttt{Subs}$: Then $m \notin s(j).\texttt{Subs}$ too as `FForward` does not affect subscriptions.

$$\begin{aligned}
&\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},\\
&\quad s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(i,e,u) \wedge e \in \mathcal{P}(s) \wedge e \neq m \wedge e.\texttt{Origin} \neq i\},\\
&\quad s(j).\texttt{Pubs},\\
&\quad (\underline{\text{if}}\ (j = i \vee m \in s(j).\texttt{Seen})\ \underline{\text{then}}\ (s(j).\texttt{Seen} \setminus \mathcal{P}(s)) \cup \{m\}\\
&\qquad \underline{\text{else}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s))\rangle\ \underline{\text{if}}\ s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}\\
&\qquad\quad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle\ \underline{\text{otherwise}}\rangle
\end{aligned}$$

$\equiv\ m \in u(j).\texttt{Seen} \Rightarrow (j = i \vee m \in s(j).\texttt{Seen})$, Simplify

$$\begin{aligned}
&\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},\\
&\quad s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(i,e,u) \wedge e \in \mathcal{P}(s) \wedge e \neq m \wedge e.\texttt{Origin} \neq i\},\\
&\quad s(j).\texttt{Pubs},\\
&\quad s(j).\texttt{Seen} \setminus \mathcal{P}(s) \cup \{m\}\ \rangle\ \underline{\text{if}}\ s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}\\
&\qquad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle\ \underline{\text{otherwise}}\rangle
\end{aligned}$$

$\equiv$ Def. `f2b`, Algorithm 1, Def. `Broadcast`, Def. `BUnsubscribe`,

v

For the nodes under consideration, Algorithm 1 applies `Broadcast` followed by `BUnsubscribe` from $m.\texttt{Topic}$. If $\neg u(m.\texttt{Origin}).\texttt{Act}$, a `BLeave` step follows.

– Let $m \notin u(j).\texttt{Seen} \wedge m.\texttt{Topic} \in u(j).\texttt{Subs}$. Then $m \in s(j).\texttt{Subs}$ too as `FForward` does not affect subscriptions.

$$\begin{aligned}
&\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},\\
&\qquad s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(i,e,u) \wedge e \in \mathcal{P}(s) \wedge e.\texttt{Origin} \neq i\},\\
&\qquad s(j).\texttt{Pubs},\\
&\qquad (\underline{\text{if}}\ (j = i \vee m \in s(j).\texttt{Seen})\ \underline{\text{then}}\ (s(j).\texttt{Seen} \setminus \mathcal{P}(s)) \cup \{m\}\\
&\qquad\quad \underline{\text{else}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s))\rangle\ \underline{\text{if}}\ s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}\\
&\qquad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle\ \underline{\text{otherwise}}\rangle
\end{aligned}$$

$\equiv m \notin u(j).\texttt{Seen} \Rightarrow \neg(j = i \vee m \in s(j).\texttt{Seen})$, Simplify

$\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid \mathit{hasMsgp}(i, e, u) \wedge e \in \mathcal{P}(s) \wedge e.\mathtt{Origin} \neq i\},$
$s(j).\mathtt{Pubs},$
$s(j).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $s(j).\mathtt{Act} \vee j \in \{e.\mathtt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

$\equiv$ Algorithm 1

v

For the nodes under consideration, Algorithm 1 applies `BUnsubscribe` from $m.\mathtt{Topic}$ before it applies `Broadcast`, so that these nodes do not receive m. If $\neg u(m.\mathtt{Origin}).\mathtt{Act}$, a `BLeave` step follows.

– Let $m \notin u(j).\mathtt{Seen} \wedge m.\mathtt{Topic} \notin u(j).\mathtt{Subs}$: Then $m \notin s(j).\mathtt{Subs}$ too as `FForward` does not affect subscriptions.

$\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid \mathit{hasMsgp}(i, e, u) \wedge e \in \mathcal{P}(s) \wedge e \neq m \wedge e.\mathtt{Origin} \neq i\},$
$s(j).\mathtt{Pubs},$
$(\underline{\text{if}}\ (j = i \vee m \in s(j).\mathtt{Seen})\ \underline{\text{then}}\ (s(j).\mathtt{Seen} \setminus \mathcal{P}(s)) \cup \{m\}$
$\underline{\text{else}}\ s(j).\mathtt{Seen} \setminus \mathcal{P}(s))\rangle$ $\underline{\text{if}}$ $s(j).\mathtt{Act} \vee j \in \{e.\mathtt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

$\equiv m \notin u(j).\mathtt{Seen} \Rightarrow \neg(j = i \vee m \in s(j).\mathtt{Seen})$, Simplify

$\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \underline{\mathit{true}},$
$s(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid \mathit{hasMsgp}(i, e, u) \wedge e \in \mathcal{P}(s) \wedge e \neq m \wedge e.\mathtt{Origin} \neq i\},$
$s(j).\mathtt{Pubs},$
$s(j).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $s(j).\mathtt{Act} \vee j \in \{e.\mathtt{Origin} \mid e \neq m \wedge e \in \mathcal{P}(s)\}$
$\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$

$\equiv$ Def. `f2b`, Algorithm 1, Def. `BUnsubscribe`,

v

For the nodes under consideration, Algorithm 1 applies `BUnsubscribe`. If $\neg u(m.\mathtt{Origin}).\mathtt{Act}$, a `BLeave` step follows.

For each of the exhaustive cases of peers considered, uBv holds. Hence, uBv.

- Let $\mathit{activep}(i, s) \ \wedge \ u = \mathtt{FLeave}(s, i)$. Then,

$\mathtt{f2b}(u)$

$\equiv \mathit{Def}.\mathtt{f2b}$

$\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(u)) :: \langle \underline{\mathit{true}},$

$u(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid hasMsgp(j, e, u) \land e \in \mathcal{P}(u) \land e.\texttt{Origin} \neq j\},$
$u(j).\texttt{Pubs},$
$u(j).\texttt{Seen} \setminus \mathcal{P}(u)\rangle\rangle$ if $u(j).\texttt{Act} \lor j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(u)\}$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv \mathsf{dom}(u) = \mathsf{dom}(s),$ Lemma 3.5.1.2, $\mathsf{dom}(\texttt{f2b}(u)) = \mathsf{dom}(\texttt{f2b}(s))$

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{true},$
$u(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid hasMsgp(j, e, u) \land e \in \mathcal{P}(u) \land e.\texttt{Origin} \neq j\},$
$u(j).\texttt{Pubs},$
$u(j).\texttt{Seen} \setminus \mathcal{P}(u)\rangle\rangle$ if $u(j).\texttt{Act} \lor j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(u)\}$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv Def.\ \texttt{FLeave}$

$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{true},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid hasMsgp(j, e, s) \land e \in \mathcal{P}(u) \land e.\texttt{Origin} \neq j\},$
$s(j).\texttt{Pubs},$
$s(j).\texttt{Seen} \setminus \mathcal{P}(u)\rangle\rangle$ if $j \neq i \ \land \ (s(j).\texttt{Act} \lor j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(u)\})$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$= \psi$

We consider 2 cases:
(1) If $\mathcal{P}(u) = \mathcal{P}(s)$.

ψ
$\equiv \mathcal{P}(u) = \mathcal{P}(s)$
$\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{true},$
$s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid hasMsgp(j, e, s) \land e \in \mathcal{P}(s) \land e.\texttt{Origin} \neq j\},$
$s(j).\texttt{Pubs},$
$s(j).\texttt{Seen} \setminus \mathcal{P}(s)\rangle\rangle$ if $j \neq i \ \land \ (s(j).\texttt{Act} \lor j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(s)\})$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv Def.\texttt{f2b}, \texttt{f2b}(s) = w, Def.\texttt{BLeave},$
$\texttt{BLeave}(w, i)$

Let $v = \texttt{BLeave}(w, i) = \texttt{f2b}(u)$. Hence uBv.

(2) Otherwise $\mathcal{P}(u) \neq \mathcal{P}(s)$, only possible due to the set of messages M exclusive to $s(i).\texttt{Pdg}$ disappearing from the network, such that $\mathcal{P}(u) = \mathcal{P}(s) \setminus M$. Since $\mathcal{M}_{\texttt{Floodsub}}$ satisfies *TConnectedp*, it is not possible for messages in M to have disappeared without reaching every eligible peer, which means that they already have been delivered to each

of their eligible peers.

$$
\begin{array}{l}
\quad \psi \\
\equiv \mathcal{P}(u) = \mathcal{P}(s) \setminus M \\
\quad \langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}}, \\
\qquad\qquad s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \wedge (e \in \mathcal{P}(s) \setminus M) \wedge e.\texttt{Origin} \neq j\}, \\
\qquad\qquad s(j).\texttt{Pubs}, \\
\qquad\qquad s(j).\texttt{Seen} \setminus (\mathcal{P}(s) \setminus M)\rangle \ \underline{\text{if}}\ j \neq i \ \wedge \ (s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid e \in \mathcal{P}(s) \setminus M\}) \\
\qquad\qquad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle \ \underline{\text{otherwise}}\rangle
\end{array}
$$

$$
\begin{array}{l}
\equiv \text{ Simplify} \\
\quad \langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}}, \\
\qquad\qquad s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \wedge (e \in \mathcal{P}(s)) \ \wedge \ (e \notin M) \wedge e.\texttt{Origin} \neq j\}, \\
\qquad\qquad s(j).\texttt{Pubs}, \\
\qquad\qquad \langle \forall m \in M :: \underline{\text{if}}\ m \in s(j).\texttt{Seen}\ \underline{\text{then}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s) \cup \{m\} \\
\qquad\qquad\qquad \underline{\text{else}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s)\rangle\rangle \\
\qquad\quad \underline{\text{if}}\ j \neq i \ \wedge \ (s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid (e \in \mathcal{P}(s)) \ \wedge \ (e \notin M)\}) \\
\qquad\quad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle \ \underline{\text{otherwise}}\rangle \\
= \delta
\end{array}
$$

For the leaving peer i, we have

$$
\begin{array}{l}
\quad \delta(i) \\
\equiv j = i \\
\quad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle
\end{array}
$$

For the rest of the peers $j \neq i$, we have

$$
\begin{array}{l}
\quad \delta(j) \\
\equiv j \neq i \\
\ \langle \underline{\mathit{true}}, \\
\quad s(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \wedge (e \in \mathcal{P}(s)) \ \wedge \ (e \notin M) \wedge e.\texttt{Origin} \neq j\}, \\
\quad s(j).\texttt{Pubs}, \\
\quad \langle \forall m \in M :: \underline{\text{if}}\ (m \in s(j).\texttt{Seen})\ \underline{\text{then}}\ (s(j).\texttt{Seen} \setminus \mathcal{P}(s)) \cup \{m\} \\
\qquad\qquad\qquad \underline{\text{else}}\ s(j).\texttt{Seen} \setminus \mathcal{P}(s)\rangle \underline{\text{if}}\ s(j).\texttt{Act} \vee j \in \{e.\texttt{Origin} \mid (e \in \mathcal{P}(s)) \ \wedge \ (e \notin M)\} \\
\quad \langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing\rangle \ \underline{\text{otherwise}}\rangle
\end{array}
$$

Consider the following algorithm on the `Broadcastsub` side:

Algorithm 2 LeaveM

Require: State w, Set of Messages M, Target Configuration u, and Leaving peer i
Ensure: A state v produced after `Broadcastsub` transitions where i leaves and messages in M are broadcast
Input: w, m, u
$v \leftarrow \texttt{BLeave}(w, i)$ ▷ $activep(i, w)$ from Def. `f2b` .
$M \leftarrow \mathcal{P}(s) \setminus \mathcal{P}(u)$
for $m \in M$ **do**
$v \leftarrow$ BroadcastM(w, m, u)
end for
return v

Each step in Algorithm 2 is enabled by construction from Definition 3.4.21, and the algorithm preserves `bp` invariants. We justify guards for each of the `Broadcastsub` steps. By inspection, the algorithm is terminating as the total number of messages in set M is finite the for-loop processes each message at most once. Let v be the `Broadcastsub` state returned from running Algorithm 2 on w. We prove uBv. `f2b` relates state of i in u to the state of i in v, due to the `BLeave` step. The states of the rest of the peers $j \neq i$ is $\delta(j)$ which is exactly the state of each peer j for each $m \in M$ we saw in the `FForward` case where $\mathcal{P}(u) = \mathcal{P}(s) \setminus \{m\}$. So, Algorithm 2 repeatedly calls Algorithm 1 for each $m \in M$. For each call, we have uBv, and hence, finally we have uBv.

- Let $\neg activep(i, s) \wedge i \notin \texttt{rng}(nbrs) \wedge \ u = \texttt{FJoin}(s, i, subs, pubs, nbrs)\rangle$.

$\texttt{f2b}(u)$
$\equiv Def.\texttt{f2b}$,
$\langle \lambda j \in \texttt{dom}(\texttt{f2b}(u)) :: \langle \underline{true},$
$u(j).\texttt{Subs} \cup \{m.\texttt{Topic} \mid hasMsgp(j, m, u) \wedge m \in \mathcal{P}(u) \wedge m.\texttt{Origin} \neq j\},$
$u(j).\texttt{Pubs},$
$u(j).\texttt{Seen} \setminus \mathcal{P}(u)\rangle$ $\underline{\text{if}}$ $u(j).\texttt{Act} \vee j \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(u)\}$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$
$\equiv \mathcal{P}(u) = \mathcal{P}(s)$
$\langle \lambda j \in \texttt{dom}(\texttt{f2b}(u)) :: \langle \underline{true},$
$u(j).\texttt{Subs} \cup \{m.\texttt{Topic} \mid hasMsgp(j, m, u) \wedge m \in \mathcal{P}(s) \wedge m.\texttt{Origin} \neq j\},$
$u(j).\texttt{Pubs},$
$u(j).\texttt{Seen} \setminus \mathcal{P}(s)\rangle$ $\underline{\text{if}}$ $u(j).\texttt{Act} \vee j \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(s)\}$
$\langle \underline{false}, \varnothing, \varnothing, \varnothing\rangle$ $\underline{\text{otherwise}}\rangle$
$\equiv Def.\ \texttt{FJoin}$
$\langle \lambda j \in \texttt{dom}(\texttt{f2b}(u)) :: \langle \underline{true},$

$s(j).\mathtt{Subs} \cup \{m.\mathtt{Topic} \mid hasMsgp(j,m,s) \wedge m \in \mathcal{P}(s) \wedge m.\mathtt{Origin} \neq j\},$
$s(j).\mathtt{Pubs},$
$s(j).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ if $s(j).\mathtt{Act} \vee j = i \vee j \in \{m.\mathtt{Origin} \mid m \in \mathcal{P}(s)\}$
$\langle \mathit{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

$\equiv \mathtt{f2b}(s) = w \wedge \neg activep(i,w)$

$\mathrm{extend}(w, i, \langle \mathit{true}, subs, pubs, [\,], \varnothing\rangle)$

$\equiv Def.\ \mathtt{BJoin}$

$\mathtt{BJoin}(w, i, subs, pubs)$

Note that $\neg activep(i,w)$ holds: if $i \notin \mathsf{dom}(s)$ then $i \notin \mathsf{dom}(\mathtt{f2b}(s))$ so $i \notin \mathsf{dom}(w)$; if $i \in \mathsf{dom}(s)$ but $\neg s(i).\mathtt{Act}$ then $\mathtt{f2b}$ preserves $\mathtt{Act}$, so $\neg w(i).\mathtt{Act}$. In either case $\neg activep(i,w)$, satisfying the guard of $\mathtt{BJoin}$.

Let $v = \mathtt{BJoin}(w, i, subs, pubs)$. Hence uBv.

- Let $activep(i,s) \ \wedge \ T \in 2^{\mathcal{T}} \ \wedge \ u = \mathtt{FUnsubscribe}(s, i, T)$.

  $\mathtt{f2b}(u)$

  $\equiv \ Def.\mathtt{f2b}$

  $\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(u)) :: \langle \mathit{true},$
  $u(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(j,e,u) \wedge e \in \mathcal{P}(u) \wedge e.\mathtt{Origin} \neq j\},$
  $u(j).\mathtt{Pubs},$
  $u(j).\mathtt{Seen} \setminus \mathcal{P}(u)\rangle$ if $u(j).\mathtt{Act} \vee j \in \{m.\mathtt{Origin} \mid m \in \mathcal{P}(u)\}$
  $\langle \mathit{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

  $\equiv \mathcal{P}(u) = \mathcal{P}(s),\ \mathsf{dom}(u) = \mathsf{dom}(s),$ Lemma 3.5.1.2, $\mathsf{dom}(\mathtt{f2b}(u)) = \mathsf{dom}(\mathtt{f2b}(s))$

  $\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \mathit{true},$
  $u(j).\mathtt{Subs} \cup \{e.\mathtt{Topic} \mid hasMsgp(j,e,s) \wedge e \in \mathcal{P}(s) \wedge e.\mathtt{Origin} \neq j\},$
  $u(j).\mathtt{Pubs},$
  $u(j).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ if $u(j).\mathtt{Act} \vee j \in \{m.\mathtt{Origin} \mid m \in \mathcal{P}(s)\}$
  $\langle \mathit{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

  $\equiv Def.\mathtt{FUnsubscribe}$

  $\langle \lambda j \in \mathsf{dom}(\mathtt{f2b}(s)) :: \langle \mathit{true},$
  (if $j = i$ then $s(j).\mathtt{Subs} \setminus T$ else $s(j).\mathtt{Subs}$) $\cup$
  $\{e.\mathtt{Topic} \mid hasMsgp(j,e,s) \wedge e \in \mathcal{P}(s) \wedge e.\mathtt{Origin} \neq j\},$
  $s(j).\mathtt{Pubs},$
  $s(j).\mathtt{Seen} \setminus \mathcal{P}(s)\rangle$ if $s(j).\mathtt{Act} \vee j \in \{m.\mathtt{Origin} \mid m \in \mathcal{P}(s)\}$
  $\langle \mathit{false}, \varnothing, \varnothing, \varnothing\rangle$ otherwise$\rangle$

  $\equiv \mathtt{f2b}(s) = w, Def.\ \mathtt{f2b}, Def.\ \mathtt{BUnsubscribe}$

$\texttt{BUnsubscribe}(w, i, T \setminus \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \land e \in \mathcal{P}(s) \land e.\texttt{Origin} \neq j\})$

Let $v = \texttt{BUnsubscribe}(w, i, T \setminus \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \land e \in \mathcal{P}(s) \land e.\texttt{Origin} \neq j\}) = \texttt{f2b}(u)$. Hence uBv.

- Let $\mathit{activep}(i, s) \ \land \ T \in 2^{\mathcal{T}} \ \land \ u = \texttt{FSubscribe}(s, i, T)$.

  $\texttt{f2b}(u)$

  $\equiv$ *Def*.f2b

  $\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(u)) :: \langle \underline{\mathit{true}},$
  $u(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, u) \land e \in \mathcal{P}(u) \land e.\texttt{Origin} \neq j\},$
  $u(j).\texttt{Pubs},$
  $u(j).\texttt{Seen} \setminus \mathcal{P}(u) \rangle$ $\underline{\text{if}}$ $u(j).\texttt{Act} \lor j \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(u)\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing \rangle$ $\underline{\text{otherwise}} \rangle$

  $\equiv \mathcal{P}(u) = \mathcal{P}(s),\ \mathsf{dom}(u) = \mathsf{dom}(s),$ Lemma 3.5.1.2, $\mathsf{dom}(\texttt{f2b}(u)) = \mathsf{dom}(\texttt{f2b}(s))$

  $\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
  $u(j).\texttt{Subs} \cup \{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \land e \in \mathcal{P}(s) \land e.\texttt{Origin} \neq j\},$
  $u(j).\texttt{Pubs},$
  $u(j).\texttt{Seen} \setminus \mathcal{P}(s) \rangle$ $\underline{\text{if}}$ $u(j).\texttt{Act} \lor j \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(s)\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing \rangle$ $\underline{\text{otherwise}} \rangle$

  $\equiv$ *Def*.FSubscribe

  $\langle \lambda j \in \mathsf{dom}(\texttt{f2b}(s)) :: \langle \underline{\mathit{true}},$
  $(\underline{\text{if}}\ i = j\ \underline{\text{then}}\ s(j).\texttt{Subs} \cup T\ \underline{\text{else}}\ s(j).\texttt{Subs}) \cup$
  $\{e.\texttt{Topic} \mid \mathit{hasMsgp}(j, e, s) \land e \in \mathcal{P}(s) \land e.\texttt{Origin} \neq j\},$
  $s(j).\texttt{Pubs},$
  $s(j).\texttt{Seen} \setminus \mathcal{P}(s) \rangle$ $\underline{\text{if}}$ $s(j).\texttt{Act} \lor j \in \{m.\texttt{Origin} \mid m \in \mathcal{P}(s)\}$
  $\langle \underline{\mathit{false}}, \varnothing, \varnothing, \varnothing \rangle$ $\underline{\text{otherwise}} \rangle$

  $\equiv \texttt{f2b}(s) = w$, *Def*. f2b, *Def*. BSubscribe
  $\texttt{BSubscribe}(w, i, T)$

  Let $v = \texttt{BSubscribe}(w, i, T) = \texttt{f2b}(u)$. Hence uBv.

□

□

Having proved that Floodsub implements Broadcastsub, we can now prove the following property.

Theorem 3.5.2 (Floodsub Commitment Implies Delivery)**.** *If* $\mathcal{M}_{\texttt{Floodsub}}$ *satisfies TConnectedp, the Commitment⇒Delivery Property 2.7.14 holds for* $\mathcal{M}_{\texttt{Floodsub}}$*.*

Proof. Let $o = m.\texttt{Origin}$, $c(t) = m \in \sigma(t)(o).\mathsf{Seen}$, $e(t) = \mathsf{eligible}(i, m, \sigma(t))$ and $d(t) = m \in \sigma(t)(i).\mathsf{Seen}$. We must show that for every message m and every peer i,

$$\mathsf{AG}\Big(\big(\neg c \;\mathsf{U}\; \big(c \wedge e\big)\big) \implies \Big(\neg c \;\mathsf{U}\; \Big(c \wedge \big(\mathsf{F}(d) \vee \mathsf{F}(\neg\, e)\big)\Big)\Big)\Big)$$

holds in $\mathcal{M}_{\texttt{Floodsub}}$.

By Theorem 3.1.1, and $\mathcal{M}_{\texttt{Floodsub}}$ satisfies *TConnectedp*, the Commitment⇒Delivery property holds for $\mathcal{M}_{\texttt{Broadcastsub}}$.

By Theorem 3.5.1,

$$\mathcal{M}_{\texttt{Floodsub}} \lesssim_{\texttt{f2b}} \mathcal{M}_{\texttt{Broadcastsub}}.$$

The property is not transferred by a generic $ACTL^* \setminus X$ preservation theorem, since skipping refinement does not preserve $ACTL^* \setminus X$ in general. Instead, we argue directly that the variables appearing in the formula are invariant across the skipped steps, so the formula holds on the concrete execution whenever it holds on the abstract.

The skipped steps in the `Floodsub`-to-`Broadcastsub` refinement under `f2b` correspond to the steps of Algorithms 1 and 2, which produces the matching `Broadcastsub` state v from the `Floodsub` state w. The algorithm executes a sequence of `BUnsubscribe`, `Broadcast`, `BLeave`, `BSubscribe`, and `BUnsubscribe` steps on the abstract side. We track each variable in the formula:

- $m \in s(o).\mathsf{Seen}$: this is set to *true* by `Broadcast` and remains *true* thereafter. All steps before `Broadcast` do not touch any peer's `Seen` set, so they do not affect this variable.
- $\mathsf{eligible}(i, m)$, which depends on $\mathit{activep}(i, s)$ and $m.\texttt{Topic} \in s(i).\texttt{Subs}$:
  - The `BUnsubscribe` steps in the first loop unsubscribe peers i' with $m \notin u(i').\mathsf{Seen}$ from $m.\texttt{Topic}$, making them ineligible. However, these are precisely the peers that did *not* receive m in the concrete execution, so the formula imposes no delivery obligation on them.
  - The `Broadcast` step delivers m to all currently subscribed active peers, making $m \in s(i).\mathsf{Seen}$ true for all eligible i.
  - The `BLeave` step marks $m.\texttt{Origin}$ as inactive. This only affects eligibility of o itself, not of subscriber $i \neq o$.
  - The subsequent `BSubscribe` and `BUnsubscribe` steps restore or adjust subscriptions, but by this point m has already been delivered to all peers that were eligible at the time of `Broadcast`.
- $m \in s(i).\mathsf{Seen}$: set to *true* by `Broadcast` for all eligible peers i at that moment, and monotonically preserved thereafter since `Seen` sets only grow.

Therefore, for any peer i that is eligible at the start of the algorithms, $m \in s(i).\mathsf{Seen}$ holds after `Broadcast` and remains true throughout the rest of the algorithm. The formula is satisfied at the `Broadcast` step and all subsequent skipped steps leave the relevant variables unchanged. Hence the property is preserved from $\mathcal{M}_{\texttt{Broadcastsub}}$ to $\mathcal{M}_{\texttt{Floodsub}}$ under `f2b`. □

3.6 Discussion on the Connectedness Condition

To the best of our knowledge, no prior work formally defines or reasons about temporal connectedness as an explicit predicate on protocol executions. The works closest to ours in spirit each gesture toward connectivity over time, but treat it as an informal background assumption, a probabilistic structural property, or a convergence hypothesis, but never as a first-class formal object that can be stated, verified, and used to parameterize correctness proofs.

Leitão et al. [44] study Plumtree, a flooding-based P2P broadcast protocol that embeds a spanning tree within a gossip-based overlay. Nodes maintain two sets of peers: *eager push peers*, through which full message payloads propagate along tree branches, and *lazy push peers*, with whom nodes exchange only lightweight message identifiers (`IHAVE` messages). This lazy gossip serves as the healing mechanism: when a node fails to receive a payload along a tree branch, it detects the gap via an `IHAVE` from a lazy push peer that received the message via an alternative overlay path, and issues a `GRAFT` request to restore connectivity. Through simulations on 10,000-node networks with failure rates ranging from 10% to 95%, they empirically demonstrate that high rates of node failure can transiently break delivery paths, while failure rates below the threshold churn level sustain reliability close to 100%. However, they do not formally define the notion of temporal connectedness which is maintained below the churn threshold, and which explains the near perfect delivery rate. Instead, connectivity is treated as an informal invariant, with empirical results serving as the only evidence of its preservation.

Coretti et al. [14] design a gossip protocol for proof-of-stake blockchain protocols that is resilient to malicious participants. Their key idea is a stake-weighted random graph overlay in which each party samples a constant number of neighbors using a verifiable random function, and runs bilateral chain synchronization over these edges to propagate chains globally. Their central claim is that even after removing all malicious nodes from the overlay, the remaining honest nodes form a subgraph with diameter $O(\log n)$, which guarantees timely message delivery. However, this result is a *static, per-snapshot* graph-theoretic property: it shows the overlay is well-connected at any given point in time, but does not formally define what it means for connectivity to be *maintained across time*. As the overlay refreshes and the stake distribution evolves, temporal connectivity remains implicit, with the protocol's time-varying behavior justified informally rather than proved.

Kuhn, Lynch, and Oshman [35] study distributed computation in dynamic networks where the topology changes every round, with communication links in each round chosen by an adversary. Their central stability property, *T-interval connectivity*, requires that across every block of T consecutive rounds there exists a connected spanning subgraph that remains stable throughout, meaning every node can communicate with every other within any such window. Under this assumption they prove that any computable function of node inputs can be computed efficiently, with larger values of T yielding proportionally faster algorithms. This captures the idea that connectivity need not hold instantaneously but must recur within bounded windows. However, T-interval connectivity is a relatively coarse description of network behavior. Consider a peer p that joins after a message m has already begun propagating from its originator o. The network may remain T-interval connected throughout, and yet p may never receive m simply because it arrived after m's propagation

had passed through its neighborhood. T-interval connectivity has no way to express this: it has no notion of a specific message, its originator, or recipient eligibility. Our definition of *TConnectedp* addresses this directly—it requires a temporal path from o to p only from the moment m is committed and only while p remains eligible, cleanly handling the case where p joins too late by releasing the delivery obligation entirely.

Our work departs from all of these by making temporal connectedness a *formal predicate*. We give it a precise definition, state it explicitly as a condition on `Floodsub` executions, and parameterize our refinement proofs by it. This allows us to characterize exactly when and why correctness fails: *the negation of temporal connectedness corresponds to the existence of a message m and subscriber j such that j will never become temporally reachable from any forwarder of m*, and thus provides a concrete target for both attack mitigation and runtime monitoring. The contrast with prior work is not merely one of formality: by making the condition explicit, we expose the *minimal* connectivity assumption required for reliable flooding, rather than over-assuming a globally well-connected or probabilistically expanding network.

Chapter 4

Mechanized proof of Refinement of Floodsub

This chapter presents the mechanized refinement proof of a simplified model of Floodsub introduced in the previous chapter. In this simplified model pending messages are represented using sets rather than queues, which removes explicit ordering and scheduling behavior from the system. In doing so, it highlights the key structural differences between queue-based and set-based message handling, particularly with respect to nondeterminism and fairness assumptions. This set-based model further enriches our refinement hierarchy, providing an additional layer of abstraction that sits alongside the queue-based Floodsub model and broadens the range of implementations the hierarchy can account for. As a result, we adopt a different notion of correctness: instead of skipping refinement, we prove simulation refinement, which is stronger.

Within this setting, we develop a fully mechanized proof in ACL2s showing that Floodsub (with set-based messages) refines Broadcastsub. These proofs provide a rigorous semantic validation of the protocol structure and establish a foundation for the executable models and refinement-based analyses developed in subsequent chapters.

The corresponding mechanized models and proofs are available in the `floodsub-refinement-mechanized-proof` directory of the public artifact repository [36]. The Floodsub development comprises 476 proved theorems and approximately 10,277 lines of Lisp code.

This chapter is organized as follows. Sections 4.1 formalizes the set based message models of Broadcastsub and Floodsub in ACL2s. Section 4.2 states the refinement map and theorem. Section 4.3 explains the refinement proof structure and Section 4.4 concludes with a discussion of related work.

Chapter provenance. This chapter is based on work previously published in [38]. The presentation has been revised and extended for inclusion in this dissertation.

4.1 Model Descriptions

We model Floodsub and its specification Broadcastsub in ACL2s using transition systems consisting of states and transition relations (boolean functions on 2 states) that depend on transition functions. We call our ACL2s models Floodnet and Broadcastnet respectively. In this section we will explain each of our transition system models in a top-down fashion. The state models are self explanatory. The interesting parts of the following code listings are the transition relations, where we place conditions, not only as sanity checks or for cases, but also as guards to disallow illegal behaviour, like requiring that a peer leaving a network is already in the network. We will discuss such conditions in detail.

4.1.1 Broadcastnet

The state of a Broadcastnet is stored in a map from peers to their corresponding peer-states. We represent peers by natural numbers. A Broadcastnet peer state is a record consisting of (i) `pubs`: the set of topics in which a peer publishes, (ii) `subs`: the set of topics to which a peer subscribes to, and (iii) `seen`: the set of messages a peer has already processed. We use sorted ACL2 lists containing unique elements to represent sets. Hence, set equality is reduced to list equality. Messages are records consisting of (i) `pld`: a message payload of type string (ii) `tp`: the topic in which this message was published, and (iii) `or`: the originating peer for this message.

```
(defdata s-bn (map peer ps-bn))

(defdata-alias peer nat)

(defdata ps-bn (record
                (pubs . lot)
                (subs . lot)
                (seen . lom)))

(defdata lot (listof topic))

(defdata-alias topic var)

(defdata lom (listof mssg))

(defdata mssg (record
               (pld . string)
               (tp . topic)
               (or . peer)))
```

We will now define the transition relation for Broadcastnet. The relation `rel-step-bn` relates 2 Broadcastnet states `s` and `u` iff `s` transitions to `u`. `rel-step-bn` is an `OR` of all the possible ways `s` can transition to `u`. `rel-skip-bn` represents a transition where `s` chooses to skip, hence `u` is `s`.

```
(definec rel-step-bn (s u :s-bn) :bool
```

```
  (v (rel-skip-bn s u)
     (rel-broadcast-bn s u)
     (rel-broadcast-partial-bn s u)
     (rel-subscribe-bn s u)
     (rel-unsubscribe-bn s u)
     (rel-leave-bn s u)
     (rel-join-bn s u)))

(definecd rel-skip-bn (s u :s-bn) :bool
  (== u s))
```

`rel-broadcast-bn` defines a relation between `s` and `u`, where `u` represents the state resulting from broadcasting a message in `s`. The broadcast is modeled as an atomic operation in which all subscribers receive the message simultaneously. The function `(br-mssg-witness s u)` is a witness finding function that calculates the message that was broadcast, if one exists. Since `seen` is a set of messages implemented as an ordered list with unique elements, `br-mssg-witness` utilizes this ordering of unique messages to find the broadcasted message.

The boolean function `broadcast-bn-pre` is a conjunction of the following preconditions: (i) the broadcast message is new, *i.e.*, it is not already found in the `seen` set of any of the peers in `s`, (ii) the originating peer of the message exists in `s`, and (iii) the topic of the broadcast message is one in which the originating peer publishes messages. `broadcast-bn-pre` also appears as an input contract (`:ic`) in the definition of the `broadcast` transition function. `(== u (broadcast (br-mssg-witness s u) s))` in the definition of `rel-broadcast-bn` ensures that the message found by `br-mssg-witness` was the sole message broadcast in `s`. We use the `insert-unique` function within `broadcast-help` to add new messages while preserving order and uniqueness of the `seen` set.

In the following code snippets, we use some ACL2s syntax described as follows. `^`, `v` and `!` are macros for `and`, `or` and `not` respectively. In a `property` form, the form following the keyword `:h` is the hypothesis, while the form following the keyword `:b` is the body. `(nin a x)` stands for `(not (in a x))`. `:match` is a powerful ACL2s pattern matching capability which supports predicates, including recognizers automatically generated by `defdata`, disjunctive patterns and patterns containing arbitrary code [63]. For more expressive pattern matching, `!` is used for literal match while `&` is used as a wildcard. In a function definition form, `:ic` and `:oc` abbreviate `:input-contract` and `:output-contract` respectively.

```
(definecd rel-broadcast-bn (s u :s-bn) :bool
  (^ (br-mssg-witness s u)
     (broadcast-bn-pre (br-mssg-witness s u) s)
     (== u (broadcast (br-mssg-witness s u) s))))

(definec broadcast-bn-pre (m :mssg s :s-bn) :bool
  (b* ((origin (mget :or m))
       (origin-st (mget origin s)))
    (^ (new-bn-mssgp m s)
       origin-st
       (in (mget :tp m)
           (mget :pubs origin-st)))))
```

```
(definec br-mssg-witness (s u :s-bn) :maybe-mssg
  (cond
   ((v (endp s) (endp u)) nil)
   ((== (car s) (car u)) (br-mssg-witness (cdr s) (cdr u)))
   (t (car (set-difference-equal (mget :seen (cdar u))
                                 (mget :seen (cdar s)))))))

(defdata maybe-mssg (v nil mssg))

(definecd new-bn-mssgp (m :mssg s :s-bn) :bool
  (v (endp s)
     (^ (nin m (mget :seen (cdar s)))
        (new-bn-mssgp m (cdr s)))))

(definecd broadcast (m :mssg s :s-bn) :s-bn
  :ic (broadcast-bn-pre m s)
  (broadcast-help m s))

(definecd broadcast-help (m :mssg st :s-bn) :s-bn
  (match st
    (() nil)
    (((p . pst) . rst)
     (cons `(,p . ,(if (v (in (mget :tp m) (mget :subs pst))
                          (== p (mget :or m)))
                       (mset :seen
                             (insert-unique m (mget :seen pst))
                             pst)
                     pst))
           (broadcast-help m rst)))))

(definec insert-unique (a :all x :tl) :tl
  (match x
    (() (list a))
    ((!a . &) x)
    ((e . es) (if (<< a e) (cons a x) (cons e (insert-unique a es))))))
```

We will explain `rel-broadcast-partial-bn`, and its necessity when discussing the proof of correctness later in the paper. `rel-subscribe-bn` and `rel-unsubscribe-bn` relate states `s` and `u` where `u` represents the state obtained after a peer in `s` subscribes to or unsubscribes from a set of topics, respectively. `bn-topics-witness` calculates the peer and the set of topics it subcribes to or unsubscribes from, if there exists such peer. Notice that we reuse `bn-topics-witness` in the definition of `rel-unsubscribe-bn`, with the arguments reversed, so as to find the topics that are subscribed to in `s`, but not in `u`. The calculated set of topics are unioned with or removed from the existing set of peer topic subscriptions of the calculated peer, based on whether it is subscribing or unsubscribing. The definition of `unsubscribe-bn` is analogous to that of `subscribe-bn` and is hence omitted.

```
(definecd rel-subscribe-bn (s u :s-bn) :bool
  (^ (bn-topics-witness s u)
     (mget (car (bn-topics-witness s u)) s)
     (== u (subscribe-bn (car (bn-topics-witness s u))
                         (cdr (bn-topics-witness s u))
                         s))))

(definecd rel-unsubscribe-bn (s u :s-bn) :bool
  (^ (bn-topics-witness u s)
     (mget (car (bn-topics-witness u s)) s)
     (== u (unsubscribe-bn (car (bn-topics-witness u s))
                           (cdr (bn-topics-witness u s))
                           s))))

(definec bn-topics-witness (s u :s-bn) :maybe-ptops
  (cond
   ((v (endp s) (endp u)) nil)
   ((== (car s) (car u)) (bn-topics-witness (cdr s) (cdr u)))
   ((^ (== (caar s) (caar u))
       (set-difference-equal (mget :subs (cdar u))
                             (mget :subs (cdar s))))
    (cons (caar s)
          (set-difference-equal (mget :subs (cdar u))
                                (mget :subs (cdar s)))))
   (t nil)))

(defdata maybe-ptops (v nil (cons peer lot)))

(definecd subscribe-bn (p :peer topics :lot s :s-bn) :s-bn
  :ic (mget p s)
  (let ((pst (mget p s)))
    (mset p (mset :subs (union-equal (mget :subs pst) topics) pst) s)))
```

`rel-join-bn` and `rel-leave-bn` relate states `s` and `u` where `u` is obtained after a peer joins `s` or leaves `s`, respectively. `bn-join-witness` calculates the peer and its peer-state, if there exists a peer that joins `s`. Its definition depends on the keys of our Broadcastnet state being in order, which is guaranteed by ACL2s `map`s. A peer joins a Broadcastnet state when a new default Broadcastnet peer state is set for the corresponding peer. A peer leaves a Broadcastnet state when the entry corresponding to the leaving peer is removed from the state.

```
(definecd rel-join-bn (s u :s-bn) :bool
  (^ (bn-join-witness s u)
     (b* ((p (car (bn-join-witness s u)))
          (pst (cdr (bn-join-witness s u))))
       (^ (! (mget p s))
          (== u (join-bn p (mget :pubs pst) (mget :subs pst) s))))))
```

```
(definecd rel-leave-bn (s u :s-bn) :bool
  (^ (bn-join-witness u s)
     (mget (car (bn-join-witness u s)) s)
     (== u (leave-bn (car (bn-join-witness u s)) s))))

(definec bn-join-witness (s u :s-bn) :maybe-ppsbn
  (match (list s u)
    ((() ((q . qst) . &)) `(,q . ,qst))
    ((((p . pst) . rs1) ((q . qst) . rs2))
     (cond
      ((== `(,p . ,pst) `(,q . ,qst)) (bn-join-witness rs1 rs2))
      ((!= q p) `(,q . ,qst))  ;; Joining peer found
      (t nil)))
    (& nil)))

(defdata maybe-ppsbn (v nil (cons peer ps-bn)))

(definecd join-bn (p :peer pubs subs :lot s :s-bn) :s-bn
  :ic (! (mget p s)) ;; Join only if peer does not already exist in state
  (mset p (ps-bn pubs subs '()) s))

(definecd leave-bn (p :peer s :s-bn) :s-bn
  :ic (mget p s) ;; Leave only if peer already exists in state
  (match s
    (((!p . &) . rst) rst)
    ((r . rst) (cons r (leave-bn p rst)))))
```

4.1.2 Floodnet

A Floodnet peer-state is a record consisting of sets `pubs`, `subs` and `seen` which we described previously in context of Broadcastnet peer-states. It also consists of `pending`, which is a set of messages that have not yet been processed, and `nsubs`, a map from topics to list of peers. `nsubs` stores topic subscriptions for neighboring peers.

```
(defdata s-fn (map peer ps-fn))

(defdata ps-fn
  (record (pubs . lot)
          (subs . lot)
          (nsubs . topic-lop-map)
          (pending . lom)
          (seen . lom)))
```

When a message has not been forwarded to neighboring subscribers (processed) it remains in the `pending` set. Once it is processed, it is added to the `seen` set. In our Floodnet model, `pending` and `seen` are sets of messages, instead of queues. This simplifies the model and allows us to not worry about the order in which messages are received. Related states have equal sets of `seen` messages.

We define the transition relation `rel-step-fn` which relates two Floodnet states `s` and `u` iff `s` transitions to `u`. It encodes all the possible ways `s` can transition to `u`. `rel-skip-fn` represents a transition where `s` chooses to skip, hence `u` is `s`.

```
(definec rel-step-fn (s u :s-fn) :bool
  (v (rel-skip-fn s u)
     (rel-produce-fn s u)
     (rel-forward-fn s u)
     (rel-subscribe-fn s u)
     (rel-unsubscribe-fn s u)
     (rel-leave-fn s u)
     (rel-join-fn s u)))

(definecd rel-skip-fn (s u :s-fn) :bool
  (== u s))
```

`rel-produce-fn` relates `s` and `u` where `u` represents the state obtained after a new message has been produced in `s`. The newly produced message is one of the pending messages in `u`. The boolean function `produce-fn-pre` is a conjunction of the following preconditions: (i) the produced message is new *i.e.*, it is not already found in the `seen` or `pending` sets of any of the peers in `s`, (ii) the originating peer of the message exists in `s`, and (iii) the topic of the produced message is one in which the originating peer publishes messages. The new message is added to the set of pending messages of the originating peer.

```
(definecd rel-produce-fn (s u :s-fn) :bool
  (rel-produce-help-fn s u (fn-pending-mssgs u)))

(definec fn-pending-mssgs (s :s-fn) :lom
  (match s
    (() '())
    (((& . pst) . rst) (union-set (mget :pending pst) (fn-pending-mssgs rst))))
       )

(definec rel-produce-help-fn (s u :s-fn ms :lom) :bool
  (match ms
    (() nil)
    ((m . rst) (v (^ (produce-fn-pre m s)
                     (== u (produce-fn m s)))
                  (rel-produce-help-fn s u rst)))))

(definec produce-fn-pre (m :mssg s :s-fn) :bool
  (b* ((origin (mget :or m))
       (origin-st (mget origin s)))
    (^ (new-fn-mssgp m s)
       origin-st
       (in (mget :tp m) (mget :pubs origin-st)))))

(definecd new-fn-mssgp (m :mssg s :s-fn) :bool
  (v (endp s)
```

```
     (^ (nin m (mget :seen (cdar s)))
        (nin m (mget :pending (cdar s)))
        (new-fn-mssgp m (cdr s)))))

(definecd produce-fn (m :mssg s :s-fn) :s-fn
  :ic (produce-fn-pre m s)
  (mset (mget :or m)
        (add-pending-psfn m (mget (mget :or m) s)) s))

(definecd add-pending-psfn (m :mssg pst :ps-fn) :ps-fn
  (if (v (in m (mget :pending pst))
         (in m (mget :seen pst)))
      pst
    (mset :pending (cons m (mget :pending pst)) pst)))
```

`rel-forward-fn` relates states `s` and `u` where `u` represents the state obtained after a peer in `s` forwards a pending message. Notice that any of the pending messages in `s` are eligible to be forwarded. Notice also that there can be several peers with a given message pending, and the Floodnet can take several possible transitions to states related to the current state by `rel-forward-fn`. To model this in a constructive and deterministic way, we introduce `find-forwarder` as a skolem function which returns the first peer in the state where a given message is pending. It produces a concrete peer p in the call to `forward-fn`. Its output contract (`:oc`) specifies that the message forwarding peer it returns (i) is a peer in `s`, (ii) possesses the given message in its `pending` set, and (iii) that message is not new in `s`.

The `forward-fn` transition function simultaneously updated the state of the peer that forwards the message, using `update-forwarder-fn` and updates the `pending` sets of the neighboring subscribers by inserting the forwarded message using `forward-help-fn`. Note that messages are forwarded to all the peers subscribing to the topic of the message in the `:nsubs` map. If the forwarding peer records its own subscriptions in `:nsubs`, it can lead to `rel-forward-fn` being infinitely enabled. We will ensure that a peer does not include itself in this map, by considering *good* Floodnet states later in the paper.

```
(definecd rel-forward-fn (s u :s-fn) :bool
    (rel-forward-help-fn s u (fn-pending-mssgs s)))

(definec rel-forward-help-fn (s u :s-fn ms :lom) :bool
  (match ms
    (() nil)
    ((m . rst)
     (v (^ (in m (fn-pending-mssgs s))
           (== u (forward-fn (find-forwarder s m) m s)))
        (rel-forward-help-fn s u rst)))))

(definec find-forwarder (s :s-fn m :mssg) :peer
    :ic (in m (fn-pending-mssgs s))
    :oc (^ (mget (find-forwarder s m) s)
           (in m (mget :pending (mget (find-forwarder s m) s)))
           (! (new-fn-mssgp m s)))
```

```
    (match s
      (((p . &)) p)
      (((p . pst) . rst)
       (if (in m (mget :pending pst)) p (find-forwarder rst m)))))

(definecd forward-fn (p :peer m :mssg s :s-fn) :s-fn
  :ic (^ (mget p s)
         (in m (mget :pending (mget p s))))
  (b* ((tp (mssg-tp m))
       (pst (mget p s))
       (nsubs (mget :nsubs pst))
       (fwdnbrs (mget tp nsubs)))
    (forward-help-fn (update-forwarder-fn p m s) fwdnbrs m)))

(definec update-forwarder-fn (p :peer m :mssg s :s-fn) :s-fn
  (match s
    (() '())
    (((!p . pst) . rst) (cons `(,p . ,(forwarder-new-pst pst m)) rst))
    ((r . rst) (cons r (update-forwarder-fn p m rst)))))

(definecd forwarder-new-pst (pst :ps-fn m :mssg) :ps-fn
  (mset :seen
        (insert-unique m (mget :seen pst))
        (mset :pending
              (remove-equal m (mget :pending pst))
              pst)))

(definecd forward-help-fn (s :s-fn nbrs :lop m :mssg) :s-fn
  (match s
    (() '())
    (((q . qst) . rst)
     (cons (if (in q nbrs)
               `(,q . ,(add-pending-psfn m qst))
             `(,q . ,qst))
           (forward-help-fn rst nbrs m)))))
```

`rel-subscribe-fn` and `rel-unsubscribe-fn` relate states `s` and `u` where `u` represents the state obtained after a peer in `s` subscribes to or unsubscribes from a set of topics, respectively. They are very similar to their Broadcastnet counterparts and hence we omit their definitions.

`rel-join-fn` and `rel-leave-fn` relate states `s` and `u` where `u` represents the state obtained after a peer joins `s` or leaves `s`, respectively.`fn-join-witness` calculates the peer and its peer-state, if there exists a peer that joins `s`, and is analogous to `bn-join-witness`. `rel-join-fn` requires that the joining peer (i) is not already in `s`, and (ii) does not exist in its own `:nsubs` map. The second condition is necessary to prevent peers from endlessly forwarding messages to themselves. `join-fn` depends on `new-joinee-st-fn` which returns the state for a newly joined peer, and on `set-subs-sfn` which updates the `:nsubs` map for each of the neighboring peers of the joining node. For the sake of brevity, we omit the definitions of these helper

functions. The `rel-leave-fn` relation, similar to `rel-leave-bn` requires that there is a leaving peer, as calculated by `(fn-join-witness u s)` and that it already exist in the state. Notice that there is another requirement, that a leaving peer has no pending messages. This condition allows for graceful exit of leaving peers, guaranteeing that no pending messages are lost along with them.

```
(definecd rel-join-fn (s u :s-fn) :bool
  (^ (fn-join-witness s u)
     (b* ((p (car (fn-join-witness s u)))
          (pst (cdr (fn-join-witness s u)))
          (nbrs (topic-lop-map->lop (mget :nsubs pst))))
       (^ (! (mget p s))
          (nin p nbrs)
          (== u (join-fn p (mget :pubs pst) (mget :subs pst) nbrs s))))))

(definecd join-fn (p :peer pubs subs :lot nbrs :lop s :s-fn) :s-fn
  :ic (^ (! (mget p s))
         (nin p nbrs))
  (set-subs-sfn nbrs
                subs
                p
                (mset p (new-joinee-st-fn pubs subs nbrs s) s)))

(definecd rel-leave-fn (s u :s-fn) :bool
  (^ (fn-join-witness u s)
     (mget (car (fn-join-witness u s)) s)
     (endp (mget :pending (mget (car (fn-join-witness u s)) s)))
     (== u (leave-fn (car (fn-join-witness u s)) s))))

(definecd leave-fn (p :peer s :s-fn) :s-fn
  :ic (mget p s)
  (match s
    (() '())
    (((!p . &) . rst) rst)
    ((r . rst) (cons r (leave-fn p rst)))))
```

4.2 Correctness and the Refinement Theorem

We consider simulation refinement [55] as the notion of correctness for Floodnet and show that Floodnet is a simulation refinement of Broadcastnet. The key idea of a simulation refinement is to show that every behavior of the concrete system (Floodnet) is allowed by the abstract system (Broadcastnet). If we prove a WFS refinement then we know that for any infinite computation tree starting from some Floodnet state, we can find a related computation tree in Broadcastnet after applying the refinement map. Another consequence is that we preserve any branching time properties, excluding next time, for example, all properties in $ACTL^* \setminus X$ [7].

The refinement map that we use is `f2b` shown below. It maps Floodnet states to Broadcastnet states where pending messages have not yet been broadcasted. This is called as the commitment approach to refinement, since we are mapping to states consisting of only those messages that have been fully propagated in Floodnet and are thus considered committed.

```
(definec f2b (s :s-fn) :s-bn
  (f2b-help s (fn-pending-mssgs s)))

(definec f2b-help (s :s-fn ms :lom) :s-bn
  (if (endp s)
      '()
    (cons `(,(caar s) . ,(f2b-st (cdar s) ms))
          (f2b-help (cdr s) ms))))

(definecd f2b-st (ps :ps-fn ms :lom) :ps-bn
  (ps-bn (mget :pubs ps)
         (mget :subs ps)
         (set-difference-equal (mget :seen ps) ms)))
```

To gain a better understanding of our refinement map, we examine example traces of Floodnet and Broadcastnet in Figure 4.2.1. On the left side, we have a trace of a Floodnet, consisting of 3 green colored nodes, numbered 1, 2 and 3. The node numbered 3 is connected to nodes 1 and 2. We show pending messages on the top left of a node, and seen messages on the bottom right. So, in the second Floodnet state shown, node 1 has a pending message `m`, after a `produce-fn` transition. On the right side, we have Broadcastnet states such that for each Floodnet state on the left, we have its refinement map on the right and for each transition on the left, we show a corresponding matching transition on the right.

The transitions on the Floodnet side are as follows: (i) Node 1 produces message `m`; (ii) Node 1 forwards its pending message `m` to its connected neighboring peer 3; (iii) Node 1 leaves the network (iv) Node 2 unsubscribes from `(mssg-tp m)`, which is the message topic (iv) Node 3 unsubscribes from `(mssg-tp m)`, and finally (v) Node 3 forwards `m` to node 2. Notice that `f2b` is a clear refinement map where events like joining and leaving are not masked. Hence, in the corresponding Broadcastnet states, leave and unsubscribe transitions are matched with leave and unsubscribe transitions. However, when the message `m` can no longer be forwarded, and is no longer pending, it needs to be matched by a `broadcast`. But notice that on the Broadcastnet side (a) the originating peer (Node 1) is no longer present, which is required for a `broadcast`, due to `broadcast-bn-pre`, and (b) there are no subscribers of `m` left in the network! This issue arises from the fact that broadcasting a message in Floodnet is a highly fragmented operation, taking place over several message hops during which peers are free to leave, join, subscribe or unsubscribe. With so many moving parts in the network, it becomes impossible to specify which nodes will receive the broadcasted message in the Broadcastnet under the refinement map at the time the message is produced. To solve this problem, we generalize the Broadcastnet specification by adding another transition relation: `rel-broadcast-partial-bn` which allows us to relate 2 states `s` and `u` where `u` represents the state obtained after broadcasting a message in `s`, but only partially. This relation is defined using the `broadcast-partial` transition function, which given a message and a list of peers, sends the message to those peers.

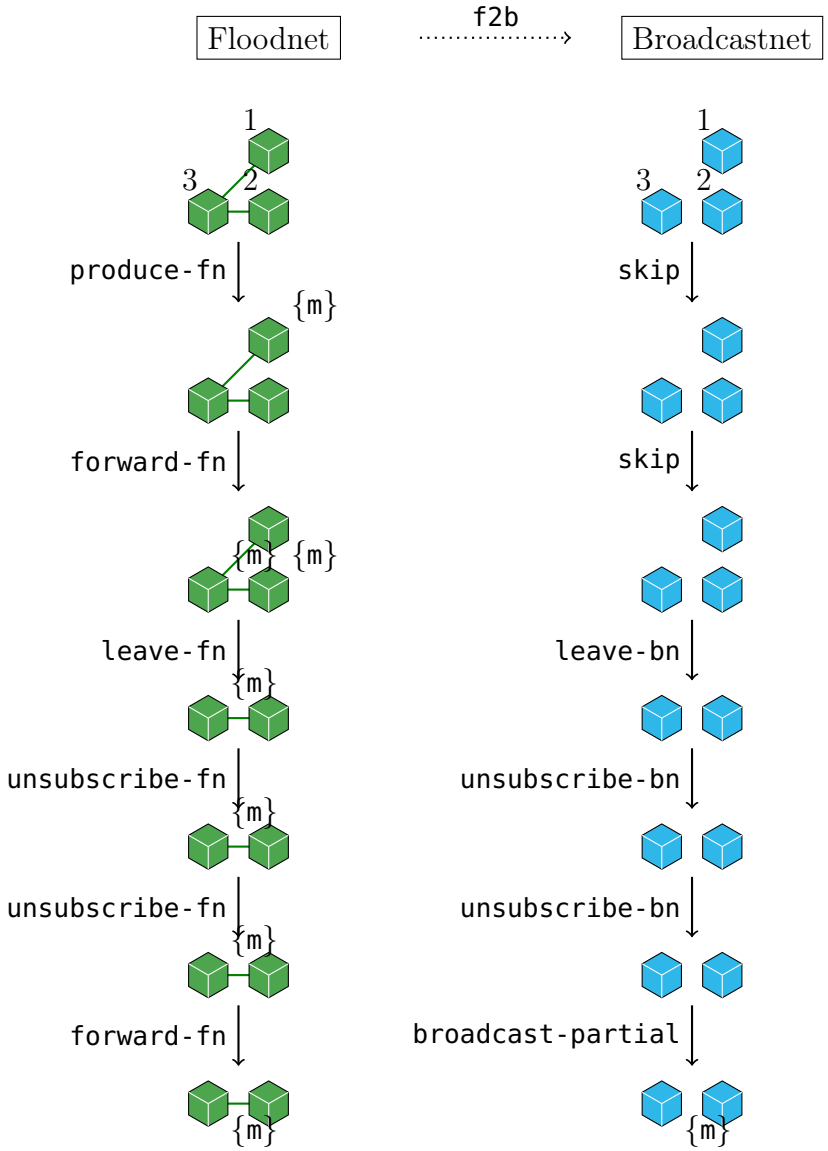


Figure 4.2.1: On the left is an example Floodnet trace.Broadcastnet states on the right are refinement maps of the Floodnet states on the left, and every step taken by the Broadcastnet states matches each step taken by the Floodnet states.

When we consider a static configuration where nodes do not change their subscriptions, no existing nodes are leaving, no new nodes are joining, and the network is connected in each topic, the recipients of the message `m` in `broadcast-partial` can be shown to be exactly those in `broadcast` *i.e.*, the subscribers of `(mssg-tp m)`. This aligns with Dijkstra's notion of self-stabilizing distributed systems [17], where the system is guaranteed to reach a legitimate configuration regardless of the initial state. In our case, once the system stabilizes (*i.e.*, peer churn and subscription/unsubscription ceases), the `broadcast-partial` step effectively becomes indistinguishable from a `broadcast` step, reinforcing the view that `broadcast-partial` is a generalization that accommodates transient perturbations while preserving desirable behavior in steady state.

```
(definecd rel-broadcast-partial-bn (s u :s-bn) :bool
  (^ (br-mssg-witness s u)
     (new-bn-mssgp (br-mssg-witness s u) s)
     (== u (broadcast-partial (br-mssg-witness s u)
                              (brd-receivers-bn (br-mssg-witness s u) u)
                              s))))

(definecd broadcast-partial (m :mssg ps :lop s :s-bn) :s-bn
  :ic (new-bn-mssgp m s)
  (broadcast-partial-help m ps s))

(definecd broadcast-partial-help (m :mssg ps :lop st :s-bn) :s-bn
  (match st
    (() nil)
    (((p . pst) . rst)
     (cons `(,p . ,(if (== p (car ps))
                       (mset :seen (insert-unique m (mget :seen pst)) pst)
                     pst))
           (broadcast-partial-help m (if (== p (car ps)) (cdr ps) ps) rst)))))
;; broadcast message receivers in a Broadcastnetwork
(definec brd-receivers-bn (m :mssg s :s-bn) :lop
  (match s
    (() ())
    (((p . pst) . rst) (if (in m (mget :seen pst))
                           (cons p (brd-receivers-bn m rst))
                         (brd-receivers-bn m rst)))))
```

We define `rel-B`, which holds for related states. `rel-B` is defined over a set of states combining both Floodnet and Broadcastnet states, which we define as `borf`. Notice that `rel-B` depends on Floodnet states satisfying the `good-s-fnp` predicate. This predicate ensures that each Floodnet state in the trace satisfies certain invariants. Given space constraints, we only list the invariant properties that hold true for `good-s-fnp` states at the end of the following listing.

```
(defdata borf (v s-bn s-fn))

(definec rel-B (x y :borf) :bool
  (v (rel-wf x y)
     (== x y)))

(definec rel-wf (x y :borf) :bool
  (^ (s-fnp x)
     (s-bnp y)
     (good-s-fnp x)
     (== y (f2b x))))

(definec rel-> (s u :borf) :bool
  (v (^ (s-fnp s) (s-fnp u) (good-rel-step-fn s u))
```

```
    (^ (s-bnp s) (s-bnp u) (rel-step-bn s u))))

(definec good-rel-step-fn (s u :s-fn) :bool
  (^ (good-s-fnp s)
     (good-s-fnp u)
     (rel-step-fn s u)))

;; A good-s-fnp state satisfies 2 predicates
(definec good-s-fnp (s :s-fn) :bool
  (^ (p!in-nsubs-s-fn s) (ordered-seenp s)))

;; Invariant 1: A peer p does not track its own subscriptions in the
;; :nsubs map. So, it can not forward a message to itself.
(propertyd prop=p!in-nsubs-s-fn (p :peer tp :topic s :s-fn)
  :h (^ (mget p s) (p!in-nsubs-s-fn s))
  :b (nin p (mget tp (mget :nsubs (mget p s)))))

;; Invariant 2: :seen components of Floodnet peers are ordered.
(property prop=ordered-seenp-cdar (s :s-fn)
  :h (^ s (ordered-seenp s))
  :b (orderedp (mget :seen (cdar s))))

(definec orderedp (x :tl) :bool
  (match x
    (() t)
    ((&) t)
    ((a . (b . &)) (^ (<< a b) (orderedp (cdr x))))))
```

Proving the WFS refinement requires proving the following three theorems: (i) WFS1 states that concrete Floodnet states are related to their corresponding Broadcastnet states under the refinement map (by relation `rel-B`), (ii) WFS2 states that the labelling function labels related states equally, and (iii) WFS3 states that given related states `s` and `w`, and given `s` steps to `u` under the transition relation, there exists a state, say `v`, such that `u` is matched by a step from `w` going to `v` such that `w` is related to `v`.

```
;; WFS1
(property b-maps-f2b (s :s-fn)
  :h (good-s-fnp s)
  :b (rel-B s (f2b s)))

;; WFS2. L is the labelling functions of our combined transition system
(definec L (s :borf) :borf
  (match s
    (:s-bn s)
    (:s-fn (f2b s))))

(property wfs2 (s w :borf)
  :h (rel-B s w)
```

```
  :b (== (L s) (L w)))

 ;; WFS3
(defun-sk exists-v-wfs (s u w)
  (exists (v)
    (^ (rel-> w v)
       (rel-B u v))))

(property wfs3 (s w u :borf)
  :h (^ (rel-B s w)
        (rel-> s u))
  :b (exists-v-wfs s u w))

;; Witness generating function for v
(definec exists-v (s u w :borf) :borf
  :ic (^ (rel-B s w)
         (rel-> s u))
  (if (null s)
      (if (null u)
          nil
        (exists-nil-v u))
    (exists-cons-v s u w)))

;; when w is nil
(definec exists-nil-v (u :borf) :borf
  :ic (^ u (rel-> nil u))
  (match u
    (:s-fn (exists-v1 nil u))
    (:s-bn u)))

;; when w is not nil
(definec exists-cons-v (s u w :borf) :borf
  :ic (^ s (rel-B s w) (rel-> s u))
  (cond
   ((^ (s-bnp s) (s-bnp w)) u)
   ((^ (s-fnp s) (s-bnp w)) (exists-v1 s u))
   ((^ (s-fnp s) (s-fnp w)) u)))

;; when s and u are Floodnet states
(definec exists-v1 (s u :s-fn) :s-bn
  :ic (good-s-fnp s)
  (cond
   ((rel-skip-fn s u) (f2b s))
   ((^ (rel-forward-fn s u)
       (!= (f2b s) (f2b u)))
    (broadcast-partial (br-mssg-witness (f2b s) (f2b u))
                       (brd-receivers-bn (br-mssg-witness (f2b s) (f2b u))
```

```
                                        (f2b u))
                    (f2b s)))
  (t (f2b u))))
```

4.3 Proof Organization

Given that we define our models using transition functions from states to states, whereas the refinement theorem is expressed in terms of transition relations, proving the monolithic refinement theorem can be a daunting task. In this section, we describe how we approached the mechanization of the refinement proof. The entire codebase can be logically partitioned into four stages:

- **State models and transition functions**: State models are described using `defdata`, and transition functions on the state models are described using `definec` and appear in files `bn-trx.lisp` and `fn-trx.lisp`. Apart from the input state, functions may accept additional arguments. For example `forward-fn` 4.1.2 accepts a peer `p` along with a message `m` that `p` forwards. These functions usually have input contracts to ensure that the extra arguments satisfy certain properties, for example, `p` should be a peer in the state `s`, and it should have `m` in its `pending` set of messages.

- **Properties of functions under the refinement map**: Given that we have defined functions that accept states and output states, we then prove theorems relating Floodnet states to their corresponding Broadcastnet states under the refinement map in file `f2b-commit.lisp`. For example, here is one such theorem:

  ```
  (property prop=forward-fn (p :peer m :mssg s :s-fn)
   :h (^ (mget p s)
         (in m (mget :pending (mget p s)))
         (== (fn-pending-mssgs (forward-fn p m s))
             (fn-pending-mssgs s)))
   :b (== (f2b (forward-fn p m s))
          (f2b s)))
  ```

  Notice that these theorems still depends on variables that have not yet been skolemized, so as to ease the theorem proving process.

- **Transition relations**: We define the transitions relations for both Broadcastnet and Floodnet in `trx-rels.lisp`. Transition relations are boolean functions over two state variables, and hence, we also define witness functions for non-state variables appearing in the transition functions. In our running example, we instantiate `p` with `(find-forwarder s m)` and `m` could be any one of the pending messages in the state.

- **Combined states, transitions relations and correctness theorems**: Finally we prove the WFS theorems and their helper properties in `f2b-sim-ref.lisp`.

During development, some of the previous iterations of our model deviated from the metatheory. For example, in one iteration of the final theorem, the restrictions on the

transition relations, which serve as guards against illegal behaviors, emerged as part of the hypotheses of WFS3. It forced us to understand the nature of good states, and to derive the required hypotheses from the invariants of the good states. In an another iteration of our models, the transition relations corresponding to each of the transitions a model can make, was augmented with natural numbers, such that transitions on the Floodnet states were matched by transitions on the Broadcastnet states bearing the same number. Eventually, this arrangement seemed unnecessary because even without the natural numbers, the theorem prover was able to pick the required transition based on theorems proved on them, and because of their definitions being disabled. Hence we would recommend to stick to the metatheory when implementing proofs of refinement, and always write the top level theorems, before embarking on proving lower-level theorems.

4.4 Bibliographic Notes

Proof mechanization in context of P2P systems has been explored previously. Azmy *et al* [3] formally verified a safety property of Pastry, a P2P Distributed Hash Table (DHT), in TLA+ [43]. The safety property is that of correct delivery, which states that at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key. Their proof assumes that nodes never fail, which is a likely event in any P2P system. Zave [88] utilized the Alloy tool [25] to produce counter-examples to show that no published version of Chord is correct w.r.t. the liveness property of the Chord ring-maintenance protocol: that the protocol can eventually repair all disruptions in the ring structure, given ample time and no further disruptions while it is working. Kumar *et al* modeled the Gossipsub [81] P2P protocol in ACL2s, formulated safety properties for its scoring function and showed using counter-examples that for some applications like Ethereum which configure Gossipsub in a particular way, it is possible for Sybil nodes to violate those properties, thereby creating large scale partition or eclipse attacks on the network [40, 41]. To the best of our knowledge, none of the previous work has attempted to prove the correctness of a P2P system by showing it as a refinement of a higher level specification.

There exist several provers to formally check properties of distributed systems, such as Dafny [23], TLA+, Ivy [67] and DistAlgo [77]. They operate by reducing a given specification to a decidable logic formula expressed entirely in First Order Logic. The basic tactic involves forming a conjunction of protocol invariants, invert it, and then using an SMT solver to (possibly) search for a counterexample. The issue in such systems is a lack of expressivity, which does not allow capturing properties over infinite traces. Another issue is that our models can be arbitrary, with nodes leaving and joining and with arbitrary pending messages in transit across a network. And we are reasoning about all possible behaviors of the protocol, which could not be done if we were to be limited to a decidable fragment of logic.

We wrote our models and proved our theorems in ACL2s. The ACL2 Sedan (ACL2s) [9, 18] is an extension of the ACL2 theorem prover [30, 32, 33]. On top of the capabilities of ACL2, ACL2s provides the following: (1) A powerful type system via the `defdata` data definition framework [13] and the `definec` and `property` forms, which support typed definitions and properties. (2) Counterexample generation capability via the `cgen` framework,

which is based on the synergistic integration of theorem proving, type reasoning and testing [10–12]. (3) A powerful termination analysis based on calling-context graphs [62] and ordinals [59–61]. (4) An (optional) Eclipse IDE plugin [9]. (5) The ACL2s systems programming framework (ASPF) [84] which enables the development of tools in Common Lisp that use ACL2, ACL2s and Z3 as a service [37, 85–87].

Chapter 5

Formalization and Verification of Meshsub

In this chapter, we describe Gossipsub v1.0 [46], which we refer to as *Meshsub* throughout this dissertation. `Meshsub` extends `Floodsub` with differentiated peering types that reduce bandwidth overhead while preserving efficient message propagation. At a high level, protocols in the libp2p pubsub family organize peers into an *overlay network*, where message dissemination occurs over a logical graph of peer connections rather than the underlying physical topology. In such overlays, peers exchange application messages and control information according to protocol-defined routing rules, enabling scalable, decentralized dissemination without requiring full connectivity among all nodes. Mesh-based pubsub protocols such as Gossipsub combine structured overlay mesh construction with gossip-based metadata exchange to balance low-latency delivery and bandwidth efficiency.

Within this framework, `Meshsub` defines distinct peering types that allow neighboring peers to negotiate the nature of their connections. Broadly, peering relationships fall into two categories:

- **Full-message peerings.** In these connections, peers maintain links over which complete messages are relayed. In protocols such as Gossipsub, full-message links form a bounded-degree mesh overlay for each topic, where peers push messages eagerly to their selected neighbors, enabling low-latency dissemination across the network. To prevent excessive flooding, the outbound degree of such connections is bounded, yielding a sparsely connected overlay dedicated to full-message propagation.

- **Metadata-only peerings.** Gossip links constitute a more densely connected overlay through which peers exchange metadata about recently seen messages (*e.g.*, message identifiers) and request missing messages. Because metadata is a small fraction of a full message, this gossip exchange imposes comparatively low bandwidth overhead while improving robustness to loss and enabling peers outside the mesh to discover messages they have missed.

In addition, `Meshsub` supports *fanout* peerings, where a publishing peer maintains full-message links to a set of downstream nodes without itself subscribing to the corresponding

topic. This unidirectional peering allows messages to enter the mesh from outside peers efficiently and subsequently be propagated through the mesh overlay.

The combination of these peering types aligns with the design of gossip-based pubsub protocols: a bounded mesh overlay supports eager push of messages among a core set of peers, while a dense metadata-exchange overlay ensures wider, lazy pull recovery and awareness. This duality enables effective bandwidth management and resilient propagation in large decentralized networks.

Since peers may join or leave the network at any time, such churn can disrupt the structure of mesh overlay networks. To address this, `Meshsub` employs explicit mesh maintenance mechanisms that operate periodically to restore and preserve desired connectivity properties. These mechanisms adjust mesh membership by selectively adding and removing peers, enforcing degree bounds and repairing broken links caused by peer departures or failures. In particular, mesh maintenance is driven by recurring control actions that rebalance the overlay, ensuring that each topic mesh remains well connected and capable of supporting efficient message dissemination despite continual network dynamics.

The remainder of this chapter is organized as follows. Section 5.1 formalizes the Meshsub model. Section 5.2 states correctness conditions. Section 5.3 develops the refinement proof. Section **??** concludes with a discussion of the main theorems.

5.1 Formal Description of the Meshsub Model

A `Meshsub` state is a partial map from peers to their states, where each peer is identified by a distinct natural number. The state for a peer $i \in \mathsf{dom}(S_{\mathtt{Meshsub}})$ is a tuple

$$\langle \mathtt{Act}, \mathtt{Subs}, \mathtt{Pubs}, \mathtt{Nbrs}, \mathtt{Mesh}, \mathtt{Fanout}, \mathtt{Pdg}, \mathtt{Ctrl}, \mathtt{WReqs}, \mathtt{RSeen}, \mathtt{Seen} \rangle.$$

The components `Act`, `Subs`, `Pubs`, `Nbrs`, `Pdg`, and `Seen` have the same interpretation as in `Floodsub`. `Nbrs`, `Mesh` and `Fanout` are total maps from topics to sets of peers. In a `Meshsub` network, peers publish/forward new messages to the members of their `Mesh` if they themselves subscribe to the topic to which they are publishing. A peer can also publish to topics they are not subscribed to, in which case they will forward messages to peers in their `Fanout` map. Both `Mesh` and `Fanout` map to neighboring peers already in the `Nbrs` map. Obviously a neighboring peer can not exist in the range of both the `Mesh` and `Fanout` maps for a particular topic.

A `Meshsub` network handles two kinds of messages: (1) `Payload` messages, that carry information, just like in `Broadcastsub` and in `Floodsub`, and (2) `Ctrl` messages that are exchanged for mesh maintenance and gossip. There are four kinds of `Ctrl` messages:

- $\langle \mathtt{IHAVE}, \mathsf{hash}(m), i \rangle$: Peer i gossips $\mathsf{hash}(m)$ (for some fixed hashing function `hash`) to a subset of its metadata-peering neighbors.
- $\langle \mathtt{IWANT}, \mathsf{hash}(m), i \rangle$: If peer i detects the absence of a message with hash $\mathsf{hash}(m)$, it issues this request to retrieve the full message
- $\langle \mathtt{GRAFT}, t, i \rangle$: Peer i sends this message to a non-mesh neighboring peer in order to graft it into its mesh for topic t.

- $\langle \texttt{PRUNE}, t, i \rangle$: Peer i sends this message to a mesh peer in order to prune it from its mesh for topic t.

`Ctrl` messages are received in the `Ctrl` queue where they are processed in FIFO order. `WReqs` is a set of message hashes corresponding to outstanding `IWANT` requests. `RSeen` records recently seen messages which are emitted in gossip. It is an array of sets of `Payload` messages, where each set consists of messages received between consecutive network maintenance events, also called as Heart Beat Maintenance (HBM). The array has a fixed length of W; at every HBM, the array slides right by one position and a fresh empty set is prepended to the left.

A `Meshsub` is parameterized by the following variables:

- D: the target degree of each node in the mesh overlay,
- D_{lo}: the low-degree threshold below which mesh maintenance adds peers ($D_{lo} \leq D$),
- D_{hi}: the high-degree threshold above which mesh maintenance prunes peers ($D_{hi} \geq D$),
- D_{lazy}: the number of non-mesh neighbors selected for gossip in each heartbeat round, and
- W: the length of the recently seen window.

Typical configurations satisfy $D_{lo} < D < D_{hi}$. In practice, Ethereum's deployed configuration uses $D = 8$, $D_{lazy} = 8$, $D_{lo} = 6$, and $D_{hi} = 10$ [75, 82].

We model the behavior of `Meshsub` as a labeled transition system $\mathcal{M}_{\texttt{Meshsub}} = \langle S_{\texttt{Meshsub}}, \rightarrow_{\texttt{Meshsub}}, L_{\texttt{Meshsub}} \rangle$, where $S_{\texttt{Meshsub}}$ is the set of Meshsub states, $\rightarrow_{\texttt{Meshsub}}$ is the transition relation defined in Tables 5.1 to 5.3, The start state of a `Meshsub` network is an empty map.

Definition 5.1.23 (Labeling function for `Meshsub`)

The labeling function $L_{\texttt{Meshsub}}$ is defined as:

$$L_{\texttt{Meshsub}}(w) = \langle \lambda i \in \mathsf{dom}(w) :: \langle w(i).\texttt{Act}, w(i).\texttt{Subs}, w(i).\texttt{Pubs}, w(i).\texttt{Nbrs}, w(i).\texttt{Seen} \rangle \rangle$$

That is, $L_{\texttt{Meshsub}}$ projects out the `Meshsub`-specific components `Mesh`, `Fanout`, `Ctrl`, `WReqs`, `RSeen`, and the `Pdg` queue.

We begin by defining the core message dissemination and gossip transitions of `Meshsub`. These transitions capture the fundamental behaviors through which messages are produced, forwarded, advertised, and requested among active peers. The `MProduce` rule models message publication: an active peer enqueues a newly produced `Payload` message into its pending queue, provided it can publish to the message's topic. The `MForward` rule models message forwarding: an active peer with a non-empty pending queue dequeues the front message, records it in its `RSeen` window and `Seen` set, and propagates it—either to its mesh peers for topics it subscribes to, or to its fanout peers otherwise (selecting fanout peers on demand if none have been chosen yet). The `MPldReq` rule models gossip reception: upon processing a front `IHAVE` control message advertising some $\mathsf{hash}(m)$, a peer checks whether it has already

seen a message with that hash; if not, it records the hash as a pending want and sends an `IWANT` request to the advertising peer. Finally, the `MPldSnd` rule models gossip response: upon processing a front `IWANT` control message, a peer delivers the requested `Payload` message directly into the requester's pending queue and clears the corresponding entry from the requester's want-requests set, provided the requester is active and has not yet seen the message. Together, these rules define the end-to-end flow of messages through the overlay network, abstracting away mesh maintenance and membership dynamics, which are handled separately. $\underline{\text{select}}(k, S)$ randomly chooses $\min(k, |S|)$ elements from a set S.

Rule	Definition
`Skip` (s)	$\underline{\text{skip}}$ (s)
`MProduce`(s, m)	$activep(m.\texttt{Origin}, s) \land m.\texttt{Topic} \in s(m.\texttt{Origin}).\texttt{Pubs} \rightarrow s(m.\texttt{Origin}).\texttt{Pdg}.\underline{\text{enqueue}}(m)$
`MForward`(s, i, D)	$activep(i, s) \land \neg s(i).\texttt{Pdg}.\underline{\text{empty}} \rightarrow \underline{\text{let}}\ m = s(i).\texttt{Pdg}.\underline{\text{front}}\ \underline{\text{in}}$ $s(i).\texttt{RSeen}[0]{:}\cup{=}\{m\}\ ;\ s(i).\texttt{Pdg}.\underline{\text{dequeue}}\ ;\ s(i).\texttt{Seen}{:}\cup{=}\{m\}\ ;$ $\underline{\text{if}}\ m.\texttt{Topic} \in s(i).\texttt{Subs}\ \underline{\text{then}}$ $\langle \forall k \in s(i).\texttt{Mesh}(m.\texttt{Topic}) : activep(k, s) \land \neg hasMsgp(k, m, s) : s(k).\texttt{Pdg}.\underline{\text{enqueue}}(m) \rangle$ $\underline{\text{else}}$ $(\underline{\text{if}}\ s(i).\texttt{Fanout} = \varnothing\ \underline{\text{then}}$ $s(i).\texttt{Fanout}(m.\texttt{Topic}){:}\cup{=}\ \underline{\text{select}}(D, s(i).\texttt{Nbrs}(m.\texttt{Topic})));$ $\langle \forall k \in s(i).\texttt{Fanout}(m.\texttt{Topic}) : activep(k, s) \land \neg hasMsgp(k, m, s) : s(k).\texttt{Pdg}.\underline{\text{enqueue}}(m) \rangle$
`MPldReq`(s, j, i, m)	$activep(j, s) \land \neg s(j).\texttt{Ctrl}.\underline{\text{empty}} \land \langle \texttt{IHAVE}, \texttt{hash}(m), i \rangle = s(j).\texttt{Ctrl}.\underline{\text{front}} \rightarrow$ $s(j).\texttt{Ctrl}.\underline{\text{dequeue}}\ ;$ $\underline{\text{if}}\ \texttt{hash}(m) \notin \{\texttt{hash}(r) \mid r \in s(j).\texttt{Seen}\} \land activep(i, s)\ \underline{\text{then}}$ $s(j).\texttt{WReqs}{:}\cup{=}\{\texttt{hash}(m)\}\ ;\ s(i).\texttt{Ctrl}.\underline{\text{enqueue}}(\langle \texttt{IWANT}, \texttt{hash}(m), j \rangle)$
`MPldSnd`(s, i, j, m)	$activep(i, s) \land \neg s(i).\texttt{Ctrl}.\underline{\text{empty}} \land \langle \texttt{IWANT}, \texttt{hash}(m), j \rangle = s(i).\texttt{Ctrl}.\underline{\text{front}} \rightarrow$ $s(i).\texttt{Ctrl}.\underline{\text{dequeue}}\ ;\ (\underline{\text{if}}\ activep(j, s) \land \neg hasMsgp(j, m, s)\ \underline{\text{then}}$ $s(j).\texttt{Pdg}.\underline{\text{enqueue}}(m)\ ;\ s(j).\texttt{WReqs}\ {:}\backslash{=}\ \{\texttt{hash}(m)\})$

Table 5.1: $\rightarrow_{\texttt{Meshsub}}$ definition, Part 1

We next define transitions governing peer membership, subscription management, and mesh maintenance. The `MGraft` rule models graft processing: upon receiving a `GRAFT` control message for topic t from peer i, a peer adds i to its mesh for t if it subscribes to t; otherwise, it responds with a `PRUNE` message to reject the graft. The `MPrune` rule models prune processing: upon receiving a `PRUNE` control message, a peer simply removes the sender from its mesh for the relevant topic. The `MSubscribe` rule models topic subscription: an active peer adds a set of topics T to its subscriptions, removes any corresponding fanout entries,

updates the neighborhood maps of all existing neighbors, and grafts into a mesh for each new topic by promoting fanout peers and selecting additional neighbors as needed, sending `GRAFT` messages to newly added mesh peers. The `MUnsubscribe` rule models topic unsubscription: an active peer removes topics T from its subscriptions, tears down the corresponding mesh entries, updates neighbors' maps, and notifies all affected mesh peers with `PRUNE` messages. The `MJoin` rule models a peer joining the network: provided the peer is not already active and does not appear in any neighbor list, it registers itself in the neighborhood maps of relevant peers, sends `GRAFT` messages to its initial mesh peers, and initializes its local state. The `MLeave` rule models a peer departing the network: an active peer removes itself from the neighbor, mesh, and fanout sets of all its neighbors for every subscribed topic, cancels all outstanding `Ctrl` messages directed to it from other peers, and marks itself as inactive. This models the real protocol behavior where disconnection drops the underlying network connection, causing any in-flight control messages addressed to the departing peer to be lost in transit. The specification states explicitly: "Whenever a peer disconnects, it is removed from the overlay". [80]. When the same peer reconnects, it establishes a completely fresh connection with no memory of the previous session. `MReset` models a hard leave, where all information about the peer is lost. Finally, the `MHBM` rule models the Heartbeat Maintenance event, which composes three sub-procedures: gossip dissemination (`MGossip`), fanout maintenance (`MFanoutMnt`), and mesh maintenance (`MMeshMnt`), each parameterized by the peer's current state and relevant configuration parameters. In contrast to the previous transitions, which focus on message flow, these rules are responsible for constructing and repairing the mesh overlay in the presence of churn, enforcing degree constraints, and ensuring that topic-specific meshes remain sufficiently connected over time.

$\mathsf{MGraft}(s, j, i)$	$activep(j, s) \ \wedge \ \neg s(j).\mathsf{Ctrl}.\underline{\text{empty}} \ \wedge \ \langle \mathsf{GRAFT}, t, i\rangle = s(j).\mathsf{Ctrl}.\underline{\text{front}} \rightarrow$ $s(j).\mathsf{Ctrl}.\underline{\text{dequeue}} \ ⨟$ $\underline{\text{if}} \ t \in s(j).\mathsf{Subs} \ \underline{\text{then}} \ s(j).\mathsf{Mesh}(t) :\cup= \{i\}$ $\underline{\text{else if}} \ activep(i, s) \ \underline{\text{then}} \ s(i).\mathsf{Ctrl}.\underline{\text{enqueue}}(\langle \mathsf{PRUNE}, t, j\rangle)$
$\mathsf{MPrune}(s, j, i)$	$activep(j, s) \ \wedge \ \neg s(j).\mathsf{Ctrl}.\underline{\text{empty}} \ \wedge \ \langle \mathsf{PRUNE}, t, i\rangle = s(j).\mathsf{Ctrl}.\underline{\text{front}} \rightarrow$ $s(j).\mathsf{Ctrl}.\underline{\text{dequeue}} \ ⨟ \ s(j).\mathsf{Mesh}(t) \ :\models \ \{i\}$
$\mathsf{MSubscribe}(s, i, T, D)$	$activep(i, s) \rightarrow \ s(i).\mathsf{Subs}:\cup=T \ ⨟ \ \langle \forall t \in T :: s(i).\mathsf{Fanout}.\mathsf{removeKey}(t)\rangle ⨟$ $\langle \forall t \in T :: \langle \forall k \in \mathsf{rng}(s(i).\mathsf{Nbrs}) :: s(k).\mathsf{Nbrs}(t):\cup=\{i\}\rangle\rangle \ ⨟$ $\langle \forall t \in T :: \underline{\text{let}} \ A =$ $\underline{\text{select}}(D - \lvert s(i).\mathsf{Fanout}(t)\rvert, s(i).\mathsf{Nbrs}(t) \setminus s(i).\mathsf{Fanout}(t)) \ \underline{\text{in}}$ $s(i).\mathsf{Mesh}(t):\cup=s(i).\mathsf{Fanout}(t) \ \uplus A ⨟$ $\langle \forall a \in A :: s(a).\mathsf{Ctrl}:\cup=\langle \mathsf{GRAFT}, t, i\rangle\rangle\rangle$
$\mathsf{MUnsubscribe}(s, i, T)$	$activep(i, s) \rightarrow s(i).\mathsf{Subs} :\models T \ \wedge \langle \forall t \in T :: s(i).\mathsf{Mesh}.\mathsf{removeKey}(t)\rangle ⨟$ $\langle \forall t \in T :: \langle \forall k \in \mathsf{rng}(s(i).\mathsf{Nbrs}) :: s(k).\mathsf{Nbrs}(t) :\models \{i\}\rangle\rangle \ \wedge$ $\langle \forall t \in T :: \langle \forall k \in s(i).\mathsf{Mesh}(t) :: s(k).\mathsf{Ctrl}:\cup=\{\langle \mathsf{PRUNE}, t, i\rangle\}\rangle\rangle$
$\mathsf{MJoin}(s, i, subs,$ $pubs, nbrs, mesh)$	$\neg activep(i, s) \ \wedge \ i \notin \mathsf{rng}(nbrs) \rightarrow$ $\langle \forall t, k \ : \ t \in subs \cup pubs \wedge \ k \in nbrs(t) \ : \ s(k).\mathsf{Nbrs}(t):\cup=\{i\}\rangle ⨟$ $\langle \forall t, k \ : \ t \in \mathsf{dom}^+(s(i).\mathsf{Mesh}) \wedge \ k \in s(i).\mathsf{Mesh}(t) \ : s(k).\mathsf{Ctrl}.\underline{\text{enqueue}}(\langle \mathsf{GRAFT}, t, i\rangle)\rangle ⨟$ $s(i) := \langle \underline{true}, subs, pubs, nbrs, mesh, [\,], \varnothing, \varnothing, \varnothing, \varnothing\rangle$
$\mathsf{MLeave}(s, i)$	$activep(i, s) \rightarrow \langle \forall t, k \ : \ t \in \mathcal{T} \wedge \ k \in s(i).\mathsf{Nbrs}(t) \ :$ $s(k).\mathsf{Nbrs}(t) :\models \{i\} ⨟ \ s(k).\mathsf{Mesh}(t) :\models \{i\} \ ⨟ \ s(k).\mathsf{Fanout}(t) :\models \{i\}\rangle \ ⨟ \langle \forall k \in \mathsf{dom}(s) \ :: \ s(k).\mathsf{Ctrl}.\underline{\text{filter}}(\langle \lambda c :: c \neq \langle -, -, i\rangle\rangle)\rangle \ ⨟$ $s(i).\mathsf{Act} := \underline{false}\rangle$
$\mathsf{MReset}(s, i)$	$s(i) := \langle \underline{false}, \varnothing, \varnothing, \varnothing, \varnothing, [\,], \varnothing, \varnothing, \varnothing, \varnothing\rangle$
$\mathsf{MHBM}(s, i, D, W, c)$	$activep(i, s) \rightarrow \mathsf{MGossip}(s, i, D, W) ⨟ \ \mathsf{MFanoutMnt}(s, i, D, c) ⨟$ $\mathsf{MMeshMnt}(s, i, D)$

Table 5.2: $\rightarrow_{\mathsf{Meshsub}}$ definition, Part 2

Finally, we introduce a collection of auxiliary transitions that are invoked periodically as part of the protocol's maintenance routines. These transitions formalize gossip emission, fanout adjustment, and mesh rebalancing, and are triggered by heartbeat events. The `MGossip` rule implements gossip dissemination: for every message m recorded across the W windows of a peer's recently-seen array `RSeen`, the peer selects up to D_{lazy} metadata neighbors for m's topic that lie outside both its mesh and fanout sets, and enqueues an `IHAVE` advertisement for m in each such active neighbor's control queue; the `RSeen` array is then advanced by prepending a fresh empty window and discarding the oldest one. The `MFanoutMnt` rule implements fanout maintenance: for each topic in the peer's fanout map, if the fanout entry has expired (modeled non-deterministically by flag c) it is removed entirely;

otherwise, if the current fanout set for that topic is smaller than the target degree D, the shortfall is made up by randomly selecting additional neighbors from those not already in the fanout set. Finally, the `MMeshMnt` rule implements mesh degree maintenance: for each topic in the peer's mesh, if the mesh size falls below the low-watermark D_{lo}, the peer selects enough additional neighbors to restore the degree to D and sends each a `GRAFT` message; conversely, if the mesh size exceeds the high-watermark D_{hi}, the peer randomly removes the excess peers down to D and notifies each removed peer with a `PRUNE` message. These mechanisms play a critical role in preserving the structural invariants of the overlay network and in supporting robustness against message loss, peer failures, and dynamic changes in connectivity.

Auxiliary Rule	Definition
$\texttt{MGossip}(s, i, D, W)$	$\langle \forall m \in \bigcup_{k=0,1,\ldots,W-1} s(i).\texttt{RSeen}[k] :: \langle \forall j \in \underline{\text{select}}(D_{lazy}, s(i).\texttt{Nbrs}(m.\texttt{Topic}) \setminus (s(i).\texttt{Mesh}(m.\texttt{Topic}) \cup s(i).\texttt{Fanout}(m.\texttt{Topic}))) :: \underline{\text{if}}\ activep(j,s)\ \underline{\text{then}}\ s(j).\texttt{Ctrl}.\underline{\text{enqueue}}(\langle \texttt{IHAVE}, \texttt{hash}(m), i\rangle)\rangle\rangle\ ;; s(i).\texttt{RSeen} := \varnothing :: \underline{\text{init}}(s(i).\texttt{RSeen})$
$\texttt{MFanoutMnt}(s, i, D, c)$	$\langle \forall t \in \texttt{dom}^+(s(i).\texttt{Fanout}) :: \underline{\text{if}}\ c\ \underline{\text{then}}\ s(i).\texttt{Fanout}.\texttt{removeKey}(t)\ \underline{\text{else if}}\ \lvert s(i).\texttt{Fanout}(t)\rvert < D\ \underline{\text{then let}}\ k = D - \lvert s(i).\texttt{Fanout}(t)\rvert\ \underline{\text{in}}\ s(i).\texttt{Fanout}(t) :\cup= \underline{\text{select}}(k, s(i).\texttt{Nbrs}(t) \setminus s(i).\texttt{Fanout}(t))\rangle$
$\texttt{MMeshMnt}(s, i, D, D_{hi}, D_{lo})$	$\langle \forall t \in \texttt{dom}^+(s(i).\texttt{Mesh}) :: \underline{\text{if}}\ \lvert s(i).\texttt{Mesh}(t)\rvert < D_{lo}\ \underline{\text{then}}\ \underline{\text{let}}\ R = \underline{\text{select}}(D - \lvert s(i).\texttt{Mesh}(t)\rvert, s(i).\texttt{Nbrs}(t) \setminus s(i).\texttt{Mesh}(t))\underline{\text{in}}\ s(i).\texttt{Mesh}(t) :\cup= R\ ;; \langle \forall k \in R : activep(k,s) : s(k).\texttt{Ctrl}.\underline{\text{enqueue}}(\langle \texttt{GRAFT}, t, i\rangle)\rangle$ $\underline{\text{else if}}\ \lvert s(i).\texttt{Mesh}(t)\rvert > D_{hi}\ \underline{\text{then}}\ \underline{\text{let}}\ R = \underline{\text{select}}(\lvert s(i).\texttt{Mesh}(t)\rvert - D, s(i).\texttt{Mesh}(t))\underline{\text{in}}\ s(i).\texttt{Mesh}(t) :\setminus= R\ ;; \langle \forall k \in R : activep(k,s) : s(k).\texttt{Ctrl}.\underline{\text{enqueue}}(\langle \texttt{PRUNE}, t, i\rangle)\rangle$

Table 5.3: Auxiliary Transition Definitions

Invariant 5.1.1. *The following invariants hold for* `Meshsub` *states:*

1. $\langle \forall m, k :: m \in s(k).\mathit{Pdg} \Rightarrow m \notin s(k).\mathit{Seen} \rangle$
2. $\langle \forall t \in \mathcal{T} :: \mathit{Mesh}(t) \cup \mathit{Fanout}(t) \subseteq \mathit{Nbrs}(t) \rangle$
3. $\langle \forall i \in \mathit{dom}(s) :: \mathit{dom}^+(s(i).\mathit{Fanout}) \subseteq s(i).\mathit{Pubs} \setminus s(i).\mathit{Subs} \rangle$
4. $\langle \forall i \in \mathit{dom}(s) :: \mathit{dom}^+(s(i).\mathit{Mesh}) \subseteq s(i).\mathit{Subs} \rangle$

5.2 Correctness Conditions for Meshsub

In `Meshsub`, message dissemination relies on a combination of direct forwarding through mesh and fanout connections, and indirect delivery through gossip and request mechanisms. Unlike `Floodsub`, where messages are forwarded eagerly to all neighbors, `Meshsub` employs selective forwarding to reduce bandwidth while maintaining delivery guarantees. So, we need temporal connectedness of such mesh/fanout overlays.

Definition 5.2.24 (Overlay Temporal Graph for message m)

Given a fullpath $\sigma = s_0 \to s_1 \to s_2 \to \cdots$ of `Meshsub` transitions and a message m, the overlay temporal graph ${}^{Mesh}G^m_\sigma = [\langle V_0, E_0\rangle, \langle V_1, E_1\rangle, \ldots]$ is defined as follows. At each step t, V_t contains every active peer, and E_t contains an edge $\langle k, i\rangle$ if one of the following holds at step t:

1. **`MForward` edge:** $s_t \xrightarrow{\texttt{MForward}(k,D)} s_{t+1}$, $m = s_t(k).\texttt{Pdg}.\underline{\text{front}}$, and $i \in s_t(k).\texttt{Mesh}(m.\texttt{Topic}) \cup s_t(k).\texttt{Fanout}(m.\texttt{Topic})$, and $\neg hasMsgp(i, m, s_t)$.

2. **`MHBM` edge:** $s_t \xrightarrow{\texttt{MHBM}(k,\ldots)} s_{t+1}$, $m \in s_t(k).\texttt{RSeen}$, and $\langle \texttt{IHAVE}, \texttt{hash}(m), k\rangle$ is sent to peer i at step t, and $\neg hasMsgp(i, m, s_t)$.

3. **`MPldReq` edge:** $s_t \xrightarrow{\texttt{MPldReq}(i,k,m)} s_{t+1}$, peer i sends `IWANT` for m to peer k, preceded by a `MHBM` edge $\langle k, i\rangle$ at some earlier step $t' < t$.

4. **`MPldSnd` edge:** $s_t \xrightarrow{\texttt{MPldSnd}(k,i,m)} s_{t+1}$, peer k delivers m to $i.\texttt{Pdg}$, preceded by a `MPldReq` edge $\langle i, k\rangle$ at some earlier step $t' < t$.

If no `MForward`, `MHBM`, `MPldReq`, or `MPldSnd` steps for m exist, define ${}^{Mesh}G^m_\sigma = [\langle \varnothing, \varnothing\rangle, \langle \varnothing, \varnothing\rangle, \ldots]$. If only finitely many qualifying steps exist, extend with $E_t = \varnothing$ for all sufficiently large t.

A temporal path (*Def*. 3.3.18) from peer o to peer i in ${}^{Mesh}G^m_\sigma$ (written $o \xrightarrow{{}^{Mesh}G^m_\sigma}{}^{*} i$) is a sequence of edges $\langle v_0, v_1\rangle, \langle v_1, v_2\rangle, \ldots, \langle v_{n-1}, v_n\rangle$ with $v_0 = o$, $v_n = i$, each edge of one of the four types above, and for each $0 \le i < n-1$, if the tuple $\langle v_i, v_{i+1}\rangle$ appears in one of the graphs $\langle V_p, E_p\rangle$ then $\langle v_{i+1}, v_{i+2}\rangle$ has to appear in some graph $\langle V_q, E_q\rangle$ such that $q > p$.

Definition 5.2.25 (Overlay Temporal Connectedness)

A fullpath σ of `Meshsub` satisfies $\mathit{OverlayTConnectedp}(\sigma)$ iff for every message m, originating peer $o = m.\texttt{Origin}$, and peer i, letting $c(t) = m \in \sigma(t)(o).\texttt{Seen}$ and $e(t) = \mathsf{eligible}(i, m, \sigma(t))$:

$$\big(\neg c \;\mathsf{U}\; \big(c \wedge e\big)\big) \implies \Big(\neg c \;\mathsf{U}\; \Big(c \wedge \big(o \xrightarrow{{}^{Mesh}G_{\sigma}^{m}}{}^{*}\; i \vee \mathsf{F}(\neg\, e)\big)\Big)\Big)$$

That is, if there exists a unique moment when m is forwarded at its originating peer o via `MForward` (and added to $o.\texttt{Seen}$) and peer i is eligible to receive it, then either there exists a temporal path from o to i in ${}^{Mesh}G_{\sigma}^{m}$ tracing the sequence of actual `MForward` or `MHBM`, `MPldReq`, and `MPldSnd` steps by which m propagates to i, or i eventually becomes ineligible (by departing or unsubscribing), in which case no delivery obligation applies. A `Meshsub` transition system satisfies *OverlayTConnectedp* iff every fullpath σ satisfies $\mathit{OverlayTConnectedp}(\sigma)$.

Every `Meshsub` transition system needs to satisfy *OverlayTConnectedp* in order to ensure that messages are eventually delivered to eligible peers. We call this condition **C0**.

Message dissemination in a `Meshsub` network differs from that in a `Floodsub` as it also makes use of gossiping. Although gossiping is not required if C0 is satisfied, we define the following conditions for gossiping:

(C1) Non-empty recently seen window. We require the recently seen window size to satisfy $W \geq 1$. This guarantees that every forwarded message remains eligible for gossip for at least one heartbeat round, preventing messages from being lost before lazy peers can be notified.

(C2) Positive gossip fanout. We require $D_{lazy} \geq 1$. This ensures that in every heartbeat round, at least one gossip advertisement is sent for some recently seen message m to a non-mesh neighbor, so that it can request m via an `IWANT`.

5.3 Correctness of Meshsub

Having defined the operational model of `Meshsub`, we now establish its correctness by relating it to the simpler specification `Floodsub`. Our goal is to show that `Meshsub` refines `Floodsub`: every behavior of the concrete protocol corresponds to a behavior of the abstract flooding specification under an appropriate abstraction map.

5.3.1 The Refinement Proof

We prove that `Meshsub` is a skipping simulation refinement of `Floodsub`. Let `mp` and `fp` be the recognizer functions for `Meshsub` and `Floodsub` states respectively. To show that `Meshsub` is a skipping simulation refinement of `Floodsub`, we find a relation B and show that B is a

SKS on the TS $\langle S, \to, \mathcal{L} \rangle$, where $S = S_{\texttt{Meshsub}} \uplus S_{\texttt{Floodsub}}$, $\to = \to_{\texttt{Meshsub}} \uplus \to_{\texttt{Floodsub}}$, and

$$\mathcal{L}(s) = \begin{cases} L_{\texttt{Meshsub}}(s) & \text{if } \mathsf{mp}(s) \\ L_{\texttt{Floodsub}}(s) & \text{otherwise} \end{cases}$$

Definition 5.3.26 (Gossiping Frontier)
The *gossiping frontier* of a `Meshsub` state s, written $\mathcal{G}(s)$, is defined as:

$$\mathcal{G}(s) = \bigcup_{i:\mathit{activep}(i,s)} \bigcup_{k=0}^{W-1} s(i).\mathsf{RSeen}[k]$$

That is, $\mathcal{G}(s)$ is the set of messages currently recorded in any peer's recently-seen windows, and hence being advertised via `IHAVE` gossip.

Definition 5.3.27 (Message Frontier)
The *message frontier* of a state s, written $\mathcal{F}(s)$, is defined as:

$$\mathcal{F}(s) = \begin{cases} \mathcal{P}(s) & \text{if } \mathsf{fp}(s) \\ \mathcal{P}(s) \cup \mathcal{G}(s) & \text{if } \mathsf{mp}(s) \end{cases}$$

That is, $\mathcal{F}(s)$ is the set of messages currently in transit: pending in `Pdg` queues in `Floodsub`, and additionally being gossiped via `IHAVE` in `Meshsub`.

Figure 5.3.1 shows a comparison of message frontier evolution in `Meshsub` (red) and `Floodsub` (green) for a five-peer network. The `Meshsub` satisfies conditions C0, C1 and C2. The initial topology is a C-shape (peers 2–1, 2–3, 3–4); peer 5 joins to the right of peer 1 after the first forwarding step, connected to peers 1 and 4 by non-mesh edges (dashed). In `Floodsub`, peer 5 receives m only when a neighbor fires `FForward`. In `Meshsub`, the gossiping frontier $\mathcal{G}(s)$ — shown at peer 1 after `MForward` adds m to 1.`RSeen` — exposes m to non-mesh neighbors via `IHAVE` advertisement over the non-mesh edge 1–5. Peer 5 then retrieves m directly from peer 1 via `MPldReq` / `MPldSnd`, bypassing the mesh chain entirely. This gossip shortcut illustrates how $\mathcal{G}(s)$ extends the effective frontier beyond the mesh topology, enabling faster dissemination to non-mesh peers.

5.3.2 Commitment Map

We define `m2f`, a *commitment map* that maps a `Meshsub` state to a `Floodsub` state by hiding all messages still in transit in $\mathcal{F}(s)$ and retaining only what has been fully disseminated. Since `Pdg` is always empty in `m2f`(s), the image is always a quiescent `Floodsub` state with no pending messages.

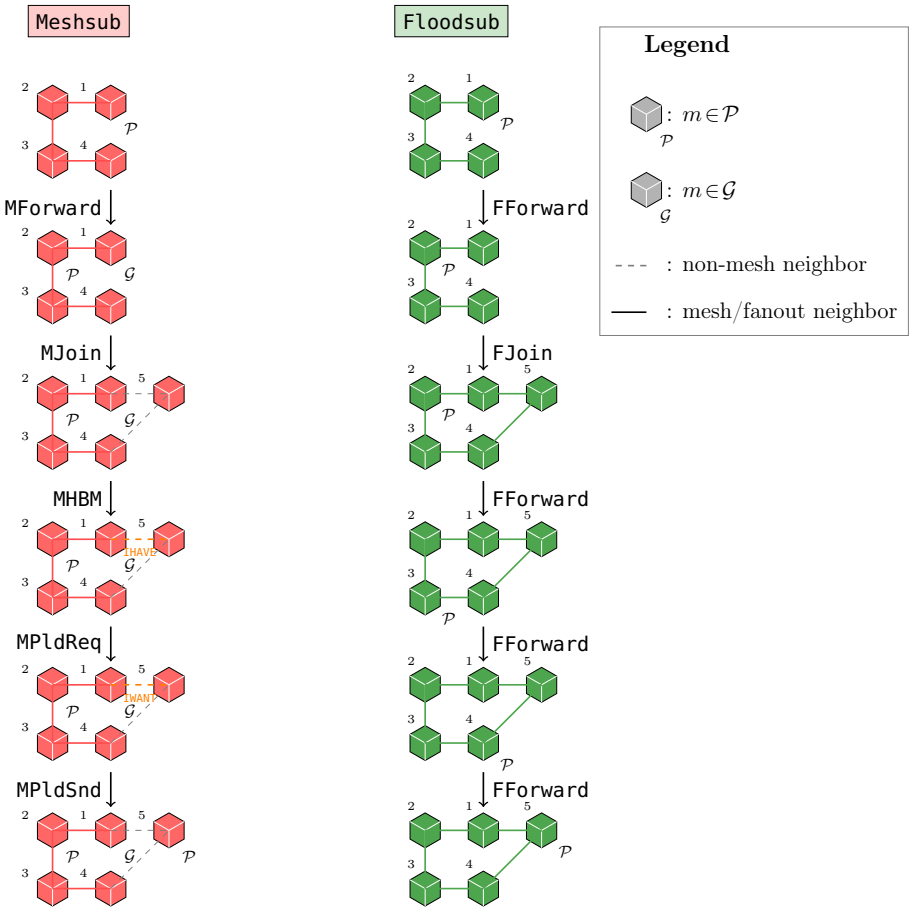


Figure 5.3.1: Evolution of the message frontier $\mathcal{F}$ in `Meshsub` (red) and `Floodsub` (green).

Definition 5.3.28 (Commitment map from `Meshsub` to `Floodsub` states)
Let $act(i,s) = s(i).\texttt{Act} \vee \bigvee_{m \in \mathcal{F}(s)} hasMsgp(i,m,s)$ denote the inflated activity condition. Then:

$$\texttt{m2f}(s) = \Big\langle \lambda i \in \mathsf{dom}(s) :: \begin{cases} \langle \underline{true}, \\ \quad s(i).\texttt{Subs} \cup \underbrace{\{m.\texttt{Topic} \mid hasMsgp(i,m,s) \wedge m \in \mathcal{F}(s) \wedge m.\texttt{Origin} \neq i\}}_{\text{subscriptions}}, \\ \quad s(i).\texttt{Pubs}, \\ \quad \underbrace{\langle \lambda \tau :: s(i).\texttt{Nbrs}(\tau) \cup \{j \mid act(j,s) \wedge i \in s(j).\texttt{Nbrs}(\tau)\}\rangle}_{\text{neighbors}}, \\ \quad [\,], \\ \quad s(i).\texttt{Seen} \setminus \mathcal{F}(s)\rangle & \underline{\text{if}}\ act(i,s) \\ \\ \langle \underline{false}, \varnothing, \varnothing, \varnothing, [\,], \varnothing\rangle & \underline{\text{otherwise}} \end{cases} \Big\rangle$$

That is, $\texttt{m2f}(s)$ constructs an `Floodsub` state where:

- A peer i is active iff $act(i,s)$ holds: it is active in s, or it holds any message in $\mathcal{F}(s)$.
- Subscriptions are inflated to include the topics of all messages that i holds in $\mathcal{F}(s)$, excluding messages originated by i itself.
- Neighbors are inflated: for each topic τ, $\texttt{m2f}(s)(i).\texttt{Nbrs}(T)$ includes all of $s(i).\texttt{Nbrs}(T)$ plus any peer j that is being made active in $\texttt{m2f}(s)$ (i.e., $act(j,s)$ holds) and has i in its own neighbor set $s(j).\texttt{Nbrs}(T)$.
- The `Pdg` queue is empty: $[\,]$.
- The seen set retains only fully delivered messages: $s(i).\texttt{Seen} \setminus \mathcal{F}(s)$.

`m2f` is structurally identical to the refinement map `f2b` defined in Chapter 3 for `Floodsub` $\rightarrow$ `Broadcastsub`, with one key difference: where `f2b` uses $\mathcal{P}(s)$ (messages pending in `Floodsub Pdg` queues) as the notion of "in transit", `m2f` uses $\mathcal{F}(s) = \mathcal{P}(s) \cup \mathcal{G}(s)$ — the full `Meshsub` frontier, which additionally includes messages being advertised via `IHAVE` gossip. This reflects the richer in-transit structure of `Meshsub`: a message is not yet committed until it has left both the `Pdg` queues *and* the recently-seen windows of all peers. Thus `m2f` hides more than `f2b`: it withholds a message from the image until the gossip phase has also completed, not just the forwarding phase.

Definition 5.3.29 (The Relation B for `Meshsub-Floodsub`)

$$sBw \iff \mathsf{mp}(s) \wedge \mathsf{fp}(w) \wedge w = \texttt{m2f}(s)$$

That is, sBw iff s is an `Meshsub` state and w is exactly the `Floodsub` state obtained by applying the commitment map `m2f` to s.

Lemma 5.3.0.1 (SKS)**.** *Under conditions (C0)–(C2), for any sBw with $\mathsf{mp}(s)$ and $\mathsf{fp}(w)$, and any fullpath σ starting from $s \in S_{\texttt{Meshsub}}$, there exists a fair fullpath δ starting from $w = \texttt{m2f}(s) \in S_{\texttt{Floodsub}}$ such that $match(B, \sigma, \delta)$.*

Proof. We perform a case analysis on `Meshsub` transitions at each step t, constructing δ inductively and maintaining $\sigma(t)\, B\, \delta(\xi.a)$, i.e., $\delta(\xi.a) = \texttt{m2f}(\sigma(t))$, at each segment boundary. Define $\xi.0 = 0$ with $\delta(\xi.0) = \texttt{m2f}(s)$.

- **`Meshsub` skip, `MPldReq`, `MGraft`, `MPrune`:** None of these modify `Seen`, `Pdg`, `RSeen`, `Act`, `Subs`, `Pubs`, or `Nbrs`. Hence $\mathcal{F}$ and $act(i, s)$ are unchanged for all i, so $\texttt{m2f}(\sigma(t+1)) = \texttt{m2f}(\sigma(t))$. `Floodsub` takes a **skip**. B maintained.

- **`MProduce`**$(\sigma(t), m)$**:** m enters $m.\texttt{Origin}.\texttt{Pdg}$, so $m \in \mathcal{F}(\sigma(t+1))$. Since `MProduce` requires $activep(m.\texttt{Origin}, \sigma(t))$, we have $act(m.\texttt{Origin}, \sigma(t)) = \underline{true}$ already; activating $m.\texttt{Origin}$ does not change. For peers $j \neq m.\texttt{Origin}$: $hasMsgp(j, m, \sigma(t+1)) = \underline{false}$ since m is fresh, so $act(j, \sigma(t+1)) = act(j, \sigma(t))$ unchanged. Subs: $m.\texttt{Origin} = i$ excludes $m.\texttt{Topic}$ inflation at $m.\texttt{Origin}$; no other peer holds m. Hence $\texttt{m2f}(\sigma(t+1)) = \texttt{m2f}(\sigma(t))$. `Floodsub` takes a **skip**. B maintained.

- **`MForward`**$(\sigma(t), i, D)$**:** m moves from $i.\texttt{Pdg}$ to $i.\texttt{Seen}$ and $i.\texttt{RSeen}[0]$, and into $j.\texttt{Pdg}$ for each mesh peer $j \in D$. Since $m \in \mathcal{G}(\sigma(t+1)) \subseteq \mathcal{F}(\sigma(t+1))$, `m2f` hides m from seen sets. Each $j \in D$ is a mesh or fanout neighbor of i, hence active in $\sigma(t)$ by the network invariant, and hence already active in $\texttt{m2f}(\sigma(t))$. After '`MForward` ', $hasMsgp(j, m, \sigma(t+1)) = \underline{true}$, so $m.\texttt{Topic}$ is added to j's inflated `Subs` in $\texttt{m2f}(\sigma(t+1))$. `Floodsub` calls $SubscribeM(\texttt{m2f}(\sigma(t)), m, D)$ (Algorithm 3) to subscribe each $j \in D$ to $m.\texttt{Topic}$. $\texttt{m2f}(\sigma(t+1))$ is reached. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

- **`MPldSnd`**$(\sigma(t), i, j, m)$**:** m moves from $\mathcal{G}(\sigma(t))$ to $j.\texttt{Pdg}$, so $m \in \mathcal{F}(\sigma(t+1))$ still. `m2f` hides m from seen sets. Peer j now has $hasMsgp(j, m, \sigma(t+1)) = \underline{true}$, so $act(j, \sigma(t+1)) = \underline{true}$. If j was previously inactive in $\texttt{m2f}(\sigma(t))$, `Floodsub` calls $SubscribeM(\texttt{m2f}(\sigma(t)), m, \{j\})$ (Algorithm 3) to subscribe j to $m.\texttt{Topic}$. $\texttt{m2f}(\sigma(t+1))$ is reached. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

- **`MHBM`**$(\sigma(t), i, D, W, c)$**:** `RSeen` windows shift at peer i. For any message m that expires from all peers' `RSeen` windows and also has $m \notin \mathcal{P}(\sigma(t+1))$, m leaves $\mathcal{F}(\sigma(t+1))$ and now appears in $\texttt{m2f}(\sigma(t+1))$. Let $s' = \sigma(t+1)$.

  Let $M = \mathcal{F}(s')$ be the frontier at the successor state $s' = \sigma(t+1)$, i.e., the set of pending messages that must still be carried in the inflated `Floodsub` image. For each $m \in M$, we argue that repeated `FForward` steps in `Floodsub` will deliver m to all eligible peers.

The path from message origin to each of the peers that receive it is induced by the `MForward` and `MPldSnd` edges in ${}^{Mesh}G^{m}_{\sigma}$: at each such forwarding edge $\langle k, j\rangle$ in `Meshsub`, the `m2f` image preserves the neighbor relation ($\sigma(t)(k).\texttt{Nbrs} \subseteq \texttt{m2f}(\sigma(t))(k).\texttt{Nbrs}$), so the entire `Meshsub` temporal path is flattened into a valid `Floodsub` temporal path in $\texttt{m2f}(\sigma(t))$. Hence each `FForward` step used by FORWARDM is enabled, and the loop terminates with every frontier message delivered to all eligible peers.

`Floodsub` then fires the following algorithm once for each such m, passing the full frontier set $M = \mathcal{F}(s')$, to transition from $\texttt{m2f}(s)$ to $\texttt{m2f}(u)$:

Algorithm 3 SubscribeM

Require: `Floodsub` state v, message m, set of peers P
Ensure: `Floodsub` state with all peers in P subscribed to $m.\texttt{Topic}$
  for all $j \in P$ **do**
    if $m.\texttt{Topic} \notin v(j).\texttt{Subs}$ **then**
      $v \leftarrow \texttt{FSubscribe}(v, j, \{m.\texttt{Topic}\})$ ▷ Subscribe j to $m.\texttt{Topic}$; enabled since *activep*(j, v) holds for all $j \in P$.
    end if
  end for
  return v

Algorithm 4 ForwardM

Require: `Meshsub` state s, message m, set of messages $M = \mathcal{F}(u)$, `Floodsub` state $w = \texttt{m2f}(s)$
Ensure: `Floodsub` state $v = \texttt{m2f}(u)$
  $v \leftarrow w$
  while $\langle \exists j :: \mathit{activep}(j, u) \wedge \mathit{mSubsp}(j, m, u) \wedge m \notin u(j).\texttt{Seen}\rangle$ **do**
    $v \leftarrow \texttt{FUnsubscribe}(v, j, \{m.\texttt{Topic}\})$ ▷ Peers that do not see m in u should not see m in v; unsubscribe them temporarily.
  end while
  $v \leftarrow \texttt{FProduce}(v, m)$ ▷ Produce m at $m.\texttt{Origin}$; enabled since *activep*$(m.\texttt{Origin}, w)$ by `m2f`.
  while $\langle \exists j :: \mathit{activep}(j, v) \wedge m \in v(j).\texttt{Pdg}\rangle$ **do**
    $v \leftarrow \texttt{FForward}(v, j)$ ▷ Forward m to all eligible peers; each step enabled by preserving a *OverlayTConnectedp* edge.
  end while
  while $\langle \exists j :: \mathit{activep}(j, u) \wedge \mathit{mSubsp}(j, m, u) \wedge m \notin u(j).\texttt{Seen}\rangle$ **do**
    $v \leftarrow \texttt{FSubscribe}(v, j, \{m.\texttt{Topic}\})$ ▷ Revert previous unsubscriptions.
  end while
  while $\langle \exists j :: \neg u(j).\texttt{Act} \wedge \neg\langle \exists m' \in M :: \mathit{hasMsgp}(j, m', v)\rangle\rangle$ **do**
    $v \leftarrow \texttt{FLeave}(v, j)$
    $v \leftarrow \texttt{FReset}(v, j)$ ▷ Only remove inflated inactive peers once they carry none of the pending messages in M.
  end while
  return v

Each step in Algorithm 4 is enabled by construction from Definition 5.3.28. Here the parameter M is the frontier $\mathcal{F}(u)$, so inactive inflated peers are removed only when they carry no frontier message. The algorithm terminates since the network is finite and each loop processes each peer at most once. After running Algorithm 4 for each $m \in M$ with the common parameter $M = \mathcal{F}(u)$, $\mathcal{F}(v) = \varnothing$ and $v = \texttt{m2f}(s')$. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

- **MJoin**$(\sigma(t), i, \ldots)$**:** Peer i becomes active with given Subs, Pubs, Nbrs. Since i was inactive, it held no frontier messages, so $act(i, \sigma(t)) = \underline{false}$. After join, $i.\texttt{Act} = \underline{true}$, so $act(i, \sigma(t+1)) = \underline{true}$. Floodsub fires FJoin$(\texttt{m2f}(\sigma(t)), i, \ldots)$ with the components prescribed by m2f (same Subs and Pubs; inflated Nbrs). $\texttt{m2f}(\sigma(t+1))$ is reached. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

- **MLeave**$(\sigma(t), i)$**:** Peer i becomes inactive; its Pdg is dropped. If i is being kept active in $\texttt{m2f}(\sigma(t))$ due to inflation (i.e., $\bigvee_{m \in \mathcal{F}(\sigma(t))} hasMsgp(i, m, \sigma(t))$), then Floodsub takes a **skip** — i must remain active in m2f until those messages leave $\mathcal{F}$. Otherwise, if i is active in $\texttt{m2f}(\sigma(t))$ only because $s(i).\texttt{Act} = \underline{true}$, Floodsub fires FLeave$(\texttt{m2f}(\sigma(t)), i)$. In either case, if any m now leaves $\mathcal{F}(\sigma(t+1))$, ForwardM is called for each such m. $\texttt{m2f}(\sigma(t+1))$ is reached. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

- **MSubscribe**$(\sigma(t), i, T, D)$**:** $i.\texttt{Subs}$ gains T; $\mathcal{F}$ and act unchanged. $\texttt{m2f}(\sigma(t+1))$ adds T to $\texttt{m2f}(\sigma(t))(i).\texttt{Subs}$. Floodsub fires FSubscribe$(\texttt{m2f}(\sigma(t)), i, T)$. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

- **MUnsubscribe**$(\sigma(t), i, T)$**:** $i.\texttt{Subs}$ loses the set of topics T. For each topic $\tau \in T$, if τ is being inflated in $\texttt{m2f}(\sigma(t))$ (i.e., $\langle \exists m \in \mathcal{F}(\sigma(t)) :: hasMsgp(i, m, \sigma(t)) \wedge m.\texttt{Topic} = \tau \rangle$), then τ must remain in $i.\texttt{Subs}$ in Floodsub until those messages leave $\mathcal{F}$. Floodsub fires FUnsubscribe$(\texttt{m2f}(\sigma(t)), i, T')$ where
  $T' = \{\tau \in T \mid \neg\langle \exists m \in \mathcal{F}(\sigma(t)) :: hasMsgp(i, m, \sigma(t)) \wedge m.\texttt{Topic} = \tau \rangle\}$ is the subset of T not currently inflated. $\texttt{m2f}(\sigma(t+1))$ is reached. $\sigma(t+1)\, B\, \delta(\xi.a+1)$.

In all cases $\delta(\xi.a) = \texttt{m2f}(\sigma(t))$ is maintained at each segment boundary. Setting π to segment boundary indices and ξ to corresponding Floodsub state indices, $corr(B, \sigma, \pi, \delta, \xi)$ holds, giving $match(B, \sigma, \delta)$. □

Theorem 5.3.1 (Meshsub is a SKS refinement of Floodsub)**.** *Under conditions (C0)–(C2), B is a skipping simulation on $\langle S_{\texttt{Meshsub}} \uplus S_{\texttt{Floodsub}}, \rightarrow_{\texttt{Meshsub}} \uplus \rightarrow_{\texttt{Floodsub}}, \mathcal{L} \rangle$, and hence $\mathcal{M}_{\texttt{Meshsub}} \lesssim_B \mathcal{M}_{\texttt{Floodsub}}$.*

Proof. We verify SKS1 and SKS2.

SKS1. Given sBw, by *Def.* B, $w = \texttt{m2f}(s)$. Since m2f projects s to w by retaining only $s(i).\texttt{Seen} \setminus \mathcal{F}(s)$ and the per-peer components Act, Subs, Pubs, Nbrs, $\mathcal{L}(s) = \mathcal{L}(w)$.

SKS2. Since B is functional — w is uniquely determined by s as $w = \texttt{m2f}(s)$ — the only case is $\mathsf{mp}(s) \wedge \mathsf{fp}(w)$, which is exactly Lemma 5.3.0.1. □

Theorem 5.3.2 (Meshsub refines Broadcastsub)**.** *Under conditions (C0)–(C2),*

$$\mathcal{M}_{\texttt{Meshsub}} \lesssim_{\texttt{f2b} \circ \texttt{m2f}} \mathcal{M}_{\texttt{Broadcastsub}}$$

Proof. By Theorem 5.3.1, $\mathcal{M}_{\texttt{Meshsub}} \lesssim_{\texttt{m2f}} \mathcal{M}_{\texttt{Floodsub}}$, where `m2f` is the commitment map (Definition 5.3.28). By Theorem 3.5.1, $\mathcal{M}_{\texttt{Floodsub}} \lesssim_{\texttt{f2b}} \mathcal{M}_{\texttt{Broadcastsub}}$. By the Compositional Skipping Refinement theorem (Theorem 2.4.2), $\mathcal{M}_{\texttt{Meshsub}} \lesssim_{\texttt{f2b}\circ\texttt{m2f}} \mathcal{M}_{\texttt{Broadcastsub}}$. □

To the best of our knowledge, this is the first refinement proof connecting a mesh-based gossip protocol to an abstract broadcast model using skipping simulation.

Theorem 5.3.3 (Meshsub Commitment Implies Delivery)**.** *Under conditions (C0)–(C2) and temporal connectedness, the Commitment⇒Delivery property holds for* $\mathcal{M}_{\texttt{Meshsub}}$.

Proof. Let $o = m.\texttt{Origin}$, $c(t) = m \in \sigma(t)(o).\texttt{Seen}$, $e(t) = \mathsf{eligible}(i, m, \sigma(t))$ and $d(t) = m \in \sigma(t)(i).\texttt{Seen}$. We must show that for every message m and every peer i,

$$\mathsf{AG}\Big(\big(\neg c \;\mathsf{U}\; (c \wedge e)\big) \implies \Big(\neg c \;\mathsf{U}\; \Big(c \wedge \big(\mathsf{F}(d) \vee \mathsf{F}(\neg\, e)\big)\Big)\Big)\Big)$$

holds for $\mathcal{M}_{\texttt{Meshsub}}$.
By Theorem 3.1.1, the Commitment⇒Delivery property holds for $\mathcal{M}_{\texttt{Broadcastsub}}$.
By Theorem 5.3.2,

$$\mathcal{M}_{\texttt{Meshsub}} \lesssim_{B;B_{\texttt{f2b}}} \mathcal{M}_{\texttt{Broadcastsub}}.$$

The property is not transferred by a generic $ACTL^* \setminus X$ preservation theorem, since skipping refinement does not preserve $ACTL^* \setminus X$ in general. Instead, we argue directly that the variables appearing in the formula are invariant across all skipped steps in both refinement stages, so the formula holds on the concrete execution whenever it holds on the abstract.

Stage 1: `Meshsub` to `Floodsub` under B. The skipped steps are `Meshsub`-internal transitions: `MHBM` (heartbeat gossip), `MPldReq` (`IHAVE` processing), and `MPldSnd` (`IWANT` delivery). None of these transitions modify $s(i).\texttt{Act}$, $s(i).\texttt{Subs}$, $s(i).\texttt{Seen}$, or $s(i).\texttt{Pdg}$ for any peer i, so $\mathsf{eligible}(i, m)$ is invariant across all skipped states, and B is maintained throughout.

Stage 2: `Floodsub` to `Broadcastsub` under `f2b`. The skipped steps correspond to Algorithm 1. For any peer i that is eligible at the start of the algorithm, $m \in s(i).\texttt{Seen}$ is set to *true* by the `Broadcast` step and is monotonically preserved thereafter, since `Seen` sets only grow. The `BUnsubscribe` steps before `Broadcast` only affect peers that did not receive m in the concrete execution, imposing no delivery obligation on them. The `BLeave` step only affects $m.\texttt{Origin}$, not subscriber $i \neq o$. The subsequent `BSubscribe` and `BUnsubscribe` steps adjust subscriptions after delivery is already complete. Hence the formula is satisfied at the `Broadcast` step and all remaining skipped steps leave the relevant variables unchanged.

Since the formula variables are invariant across all skipped steps in both stages, the Commitment⇒Delivery property holds for $\mathcal{M}_{\texttt{Meshsub}}$. □

5.4 Extension to Gossipsub

Gossipsub (v1.1) extends Meshsub by introducing a scoring mechanism to evaluate peer behavior and maintain mesh quality. Each peer maintains a score for its neighbors, computed as a weighted sum of behavioral metrics including time spent in the mesh, first message deliveries, invalid message penalties, and protocol violation penalties. Peers whose score falls below certain thresholds may be pruned from the mesh, while peers with high scores are preferentially retained or grafted. The intent is to remove misbehaving or poorly performing peers and improve reliability and efficiency of message dissemination.

The scoring function is defined as a weighted sum of components $P_1, P_2, \ldots, P_n$, where positive components reward desirable behavior and negative components penalize protocol violations. The following theorem, mechanically verified in ACL2s, characterizes the structure of the scoring function.

Theorem 5.4.1 (Score Structure)**.** *A peer's score is maximized by the positive components P_1 (time in mesh) and P_2 (first message deliveries). All other components are penalties that can only decrease the score.*

Since peers following the protocol do not trigger penalties, their scores are determined entirely by P_1 and P_2. Such peers accumulate score over time and are never pruned from the mesh due to scoring. Consequently, the scoring mechanism only introduces internal transitions — score updates and penalty-based pruning of misbehaving peers — that do not affect protocol-compliant message dissemination.

Theorem 5.4.2 (Gossipsub Refines Meshsub)**.** *Gossipsub is a stuttering bisimulation of Meshsub.*

Proof sketch. Define a relation R between Gossipsub and Meshsub states that projects away the scoring components. By Theorem 5.4.1, protocol-compliant peers are never pruned due to scoring. Every Gossipsub transition either:

1. corresponds directly to a Meshsub transition (message forwarding, subscription changes, mesh maintenance), or
2. is an internal scoring update that leaves the Meshsub projection unchanged (a stuttering step).

Since R is preserved by both cases and no observable delivery events are added or removed, R is a stuttering bisimulation. □

Combining this with the Meshsub refinement chain yields:

Corollary 5.4.1 (Gossipsub Refines Broadcastsub)**.** *Under temporal connectedness,*

$$\mathcal{M}_{\text{Gossipsub}} \approx \mathcal{M}_{\text{Meshsub}} \lesssim \mathcal{M}_{\text{Floodsub}} \lesssim \mathcal{M}_{\text{Broadcastsub}}$$

This hierarchy cleanly separates concerns: the refinement from Meshsub to Broadcastsub establishes correctness of the core dissemination mechanism, while the refinement from Gossipsub to Meshsub shows that scoring preserves this correctness. The scoring mechanism provides robustness against misbehavior without compromising the protocol's fundamental guarantees. In the next chapter, we analyze how scoring defends against malicious peers.

Chapter 6

Formalization and Property-Based Testing of Gossipsub v1.1

Gossipsub v1.1 [47] extends Meshsub with security-oriented mechanisms intended to improve robustness against adversarial behavior in permissionless networks. These extensions preserve the underlying mesh-and-gossip dissemination structure of Meshsub, while adapting peer selection and mesh maintenance decisions based on observed peer behavior. In this dissertation, we model and analyze a restricted subset of Gossipsub v1.1, focusing on those mechanisms that directly affect mesh formation and message dissemination. Our model of Gossipsub is written in ACL2s and is publicly available, open, extensible, formal, official and executable. It supports reasoning about protocol properties and components, as well as running simulations that allow precise control over the network, peers and event ordering.

Our ACL2s model for Gossipsub v1.1 is available in the repository directory `gossipsub-formal` of the public artifact repository [36]. This formalization underlies the counterexample generation and attack constructions presented in the following chapters.

At the core of Gossipsub v1.1 is a *peer scoring* mechanism, whereby each peer locally assigns scores to its neighbors based on their observed behavior during message propagation and gossip exchange. These scores influence mesh maintenance decisions, including the retention of well-behaving peers and the pruning of peers whose behavior is deemed undesirable. As a result, scoring introduces a feedback loop between message dissemination outcomes and the evolving structure of the overlay mesh.

Other features described in the Gossipsub v1.1 specification—such as outbound mesh quotas, adaptive gossip dissemination, prune peer exchange, and application-level message validation—are not modeled in our formalization. By excluding these orthogonal extensions, our analysis isolates the impact of peer scoring on the Meshsub protocol core, enabling precise reasoning about its security implications.

Our model captures the essential distinction between Meshsub and Gossipsub v1.1 relevant to this work: while Meshsub relies on randomized peer selection and static degree bounds, Gossipsub v1.1 adapts mesh membership based on peer behavior. As we demonstrate in later chapters, this behavioral adaptation—despite being intended as a security enhancement—introduces subtle interactions that can lead to invalidation of expected dissemination properties (as in Meshsub).

This chapter is organized as follows. Section 6.1 describes our formal model for Gossipsub

in ACL2s. Section 6.2 discusses and reasons about the scoring function component of Gossipsub. It defines the fundamental property which should hold for any scoring function that claims to defend against malicious peers, and then shows that it does not hold for Gossipsub v1.1. Finally Section 6.3 relates the fundamental property to the Commitment⇒Delivery property which we proved for `Broadcastsub`, `Floodsub` and `Meshsub` in the previous chapters.

Chapter provenance. This chapter is based on work previously published in [40, 41]. The presentation has been revised and extended for inclusion in this dissertation.

6.1 GossipSub ACL2s Model Description

In this section, we describe our ACL2s model, while also giving an overview of how the Gossipsub protocol works. Interested readers are encouraged to walk through our ACL2s formalization whose presentation mirrors this section [79]. Consider the mesh shown in Figure 6.1.1. Full-message payloads are forwarded on the full-message peering edges, which

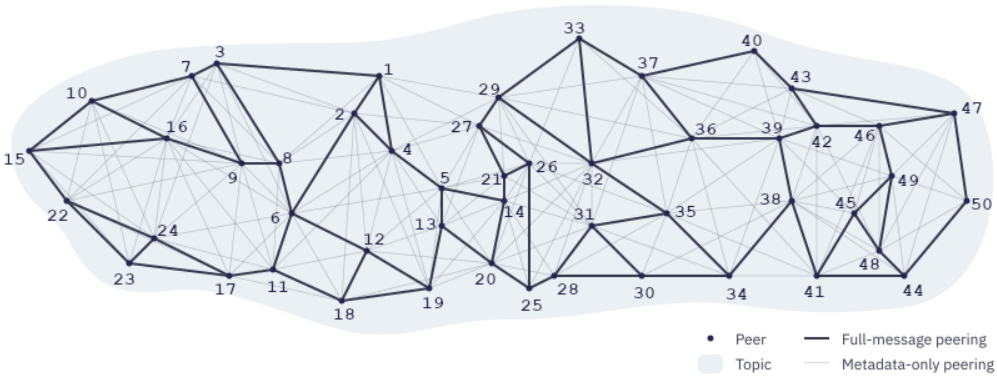


Figure 6.1.1: A mesh of peers subscribing to the same topic. This Figure was taken from [73].

consume more bandwidth, while the metadata-only peering edges are used only to advertise and request full-messages using corresponding "`IHAVE`" and "`IWANT`" Remote Procedure Calls (RPCs). These RPCs carry the metadata of the full-message being advertised or requested, which is considerably smaller than the message itself. In this way, network congestion is kept in check, and full-messages are supposed to be eventually disseminated to all peers who subscribe to the given topic. As an example, peers numbered `1` and `2` can send full-messages to each other, since they are mesh neighbors on the illustrated topic and share a full-message peering edge. However, peer `29` can receive a full-message from `1` only if it requests one, by sending an `IWANT` message in response to a corresponding `IHAVE` message received from peer `1`. Note, Figure 6.1.1 only shows a single Gossipsub mesh. However, real world applications can have an arbitrary number of topics and corresponding meshes, *e.g.*, Eth2.0 supports up to 71 topics. Figure 6.1.2 illustrates a community graph of an actual Eth2.0 network from Li *et al* [45].

In order to model and analyze Gossipsub in ACL2s, we rely heavily on Defdata to easily model network components, and on cgen for property-based testing. Distributed systems

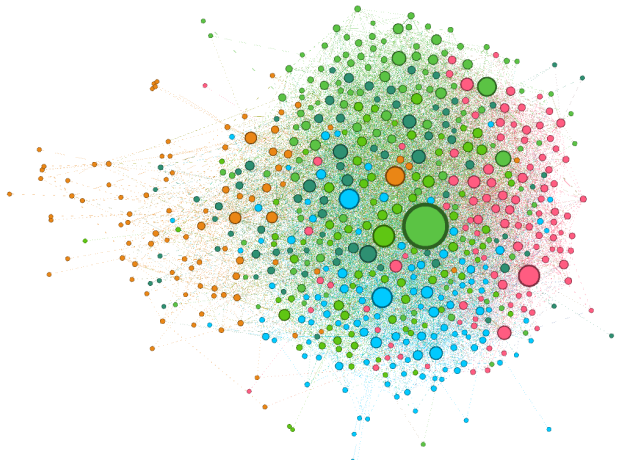


Figure 6.1.2: A community graph of Eth2.0 nodes where a bigger node implies greater degree. Similarly colored nodes have more edges among them than to the rest of the graph.

are often described in terms of *state machines*, namely automata that encode the discrete behaviors of a peer in the system [42, 76]. In our model, we describe the state-space of a Gossipsub peer with a `peer-state` type, and then implement the Gossipsub state machine using a state transition function. We use a Defdata map from peers to their corresponding states to capture the state of an entire network.

```
(defdata group (map peer peer-state))
```

We use records whenever we need to store state because of the convenience of using named fields to access the internals of the state. While proving theorems referring to states, we noticed that we routinely had to prove helper lemmas about the types of the contents of those states. This motivated us to rewrite records in the ACL2s books to enable such helper lemmas automatically, leading to cleaner code.

The local state of a peer is modeled as a record using the following definition:

```
(defdata peer-state
  (record (nts        . nbr-topic-state)
          (mst        . msgs-state)
          (nbr-tctrs  . pt-tctrs-map)
          (nbr-gctrs  . p-gctrs-map)
          (nbr-scores . peer-rational-map)))
```

where (1) `nts` is a record that stores information about the peer's neighbors' subscriptions, the peer's mesh neighbors, and the peer's fanout (which we describe later); (2) `mst` is a record that stores the peer's messages and related state; (3) `nbr-tctrs` and `nbr-gctrs` are total maps that store counters used for computing a neighbor's topic-specific and general scores (respectively); and finally (4) `nbr-scores` is a total map from peers to their scores. Note that `peer` and `topic` are ACL2 symbols, while `tctrs` is a record that keeps track of a peer's behaviors in each topic:

```
(defdata tctrs
  (record (invalidMessageDeliveries . non-neg-rational)
```

```
          (meshMessageDeliveries    . non-neg-rational)
          (meshTime                 . non-neg-rational)
          (firstMessageDeliveries   . non-neg-rational)
          (meshFailurePenalty       . non-neg-rational)))
```

Similarly, `gctrs` keeps track of a peer's general behaviors, not pertaining to any single topic:

```
(defdata gctrs
  (record (apco . rational)  ;; application provided score
          (ipco . non-neg-rational) ;; ip-colocation factor
          (bhvo . non-neg-rational))) ;; track misbehavior
```

Notice that each of the counters is a rational, not a natural. This is because counters are supposed to fractionally decay at regular intervals. The rate of decay is dependent on the application running on top of Gossipsub. We define maps for each of these counters:

```
(defdata pt (cons peer topic))
(defdata pt-tctrs-map (map pt tctrs))
(defdata p-gctrs-map (map peer gctrs))
(defdata peer-rational-map (map peer rational))
```

For each map, we define a lookup function with default values. ACL2s automatically proves termination and input-output contracts, which suffices to show that the maps are total. As an example, the following is the lookup function for scores:

```
(definecd lookup-score (p :peer prmap :peer-rational-map) :rational
  (let ((x (mget p prmap)))
    (match x
      (nil 0)  ;; default value
      (& x))))
```

We explore each component of the `peer-state`.

Neighbor topics state. A key objective of the Gossipsub protocol is to reduce network congestion, while forwarding data as quickly as possible. To achieve this, a peer forwards full messages only to a subset of its neighboring peers. A Gossipsub peer keeps track of the topics its neighbors subscribe to, using the `nbr-topicsubs` field in `nbr-topic-state`. Using this information, it is able to build up a picture of the topics around it and which peers are subscribed to each topic. The peers to which it forwards full messages in topics it does not itself subscribe to constitute a list called a *fanout*, and is stored in the `topic-fanout` field, which is a map from topics to lists of peers (`lop`). The peer forwards full messages to other peers with whom it shares mesh membership. Mesh memberships are stored in the `topic-lop-map` field `topic-mesh`. Finally, the peer stores the last time it published a message to its fanout. This is used to expire a peer's fanout, if it has been too long since the peer last published.

```
(defdata peer symbol)
(defdata lop (listof peer))
(defdata topic-lop-map (map topic lop))
(defdata topic-nnr-map (map topic non-neg-rational))
```

```
(defdata nbr-topic-state
  (record (nbr-topicsubs . topic-lop-map)
          (topic-fanout  . topic-lop-map)
          (last-pub      . topic-nnr-map)
          (topic-mesh    . topic-lop-map)))
```

Messages state. `msgs-state` is a record type used to store information about messages received, requested, seen, or forwarded by a peer. First we define the types `pid-type`, which is just an alias for the type `symbol`, and the record type `payload-type`.

```
(defdata-alias pid-type symbol)
(defdata payload-type (record (content . symbol)
                              (pid     . pid-type)
                              (top     . topic)
                              (origin  . peer)))
```

`payload-type` is a record used to represent full messages, carrying the message content, payload id, the topic of this message and the peer who originated it. pid-type represents payload ids, a hash of a message payload which can identify the full message content. The msgs-state is defined as follows:

```
(defdata msg-peer (v (cons payload-type peer)
                     (cons pid-type peer)))
(defdata msgpeer-rat (map msg-peer rational))
(defdata msgs-waiting-for (map pid-type peer))
(defdata mcache (alistof payload-type peer))

(defdata msgs-state
  (record (recently-seen   . msgpeer-rat)
          (pld-cache       . mcache)
          (hwindows        . lon)
          (waitingfor      . msgs-waiting-for)
          (served          . msgpeer-rat)
          (ihaves-received . nat)
          (ihaves-sent     . nat)))
```

`msgs-state` stores (1) `recently-seen`, a map from either a full-message or a message hash to the time since receipt; (2) `pld-cache`, an association list of full messages and their senders; (3) `hwindows`, or history windows, a list of naturals where each is the number of messages received in an interval; (4) `waitingfor`, a map from message ids of messages that haven't been received yet, to peers one has sent corresponding `IWANT` requests to; and (5) `served`, a map from either a full-message or a message hash to a rational, denoting the count of the number of times this message was served. The `served` map helps to detect peers sending too many `IWANT` messages. Finally, `msgs-state` contains (6) `ihaves-received` and `ihaves-sent`, which store the number of `IHAVE` messages received and sent, respectively.

Events. Our model includes events that can occur in a network. Events include peers sending or receiving (1) control messages like `GRAFT` or `PRUNE` for mesh control, `SUB`, `UNSUB`, `JOIN` and `LEAVE` for topics, or `CONNECT1` or `CONNECT2` messages to edit neighbors; (2) `PAYLOAD`

for carrying message payload, or `IHAVE` for advertising or `IWANT` for requesting message payloads; and (3) `HBM` for heart-beat events occurring at each peer at regular intervals. A list of events will have type `loev`. Events are defined as follows:

```
(defdata verb (enum '(SND RCV)))
(defdata rpc (v (list 'CONNECT1 lot)
                (list 'CONNECT2 lot)
                (list 'PRUNE topic)
                (list 'GRAFT topic)
                (list 'SUB topic)
                (list 'UNSUB topic)))
(defdata data (v (list 'IHAVE lopid)
                 (list 'IWANT lopid)
                 (list 'PAYLOAD payload-type)))
(defdata mssg (v rpc data))
(defdata evnt (v (cons peer (cons verb (cons peer mssg)))
                 (list peer 'JOIN topic)
                 (list peer 'LEAVE topic)
                 (list peer verb peer 'CONNECT1 lot)
                 (list peer verb peer 'CONNECT2 lot)
                 (list peer 'HBM pos-rational)
                 (list peer 'APP payload-type)))
(defdata hbm-evnt (list peer 'HBM pos-rational))
(defdata-subtype hbm-evnt evnt)

(defdata loev (listof evnt))
```

Heart-beat events occur at regular intervals at each peer in a Gossipsub network. Several maintenance activities are performed during each such event. Scores are updated for neighboring peers, which are then used to update mesh memberships. Counters are multiplied by some application-specific decay factors. Neighboring peers that are not part of any mesh are sent `GRAFT` messages at regular intervals, provided that their addition could improve the average score of peers in the corresponding mesh. Several of these actions depend on application-specific weights (used for calculating scores) and parameters.

Parameters and Scoring. We store the weights and parameters relevant for scoring in a record called a `twp`. This record totally captures the application-specific configuration of a Gossipsub instance, *e.g.*, we can uniquely specify the configuration used by Eth2.0 in a `twp`. Thus, in order to simulate an application running on top of Gossipsub with our model, all we need to know is the application-specific `twp`. This is what we mean when we say our model is "pluggable".

```
(defdata weights
  (record (w1  . non-neg-rational)
          (w2  . non-neg-rational)
          (w3  . non-pos-rational)
          (w3b . non-pos-rational)
          (w4  . neg-rational)
```

```
          (w5  . non-neg-rational)
          (w6  . neg-rational)
          (w7  . neg-rational)))

(defdata params
  (record (activationWindow               . nat)
          (meshTimeQuantum                . pos)
          (p2cap                          . nat)
          (timeQuantaInMeshCap            . nat)
          (meshMessageDeliveriesCap       . pos-rational)
          (meshMessageDeliveriesThreshold . pos-rational)
          (topiccap                       . rational)
          (grayListThreshold              . rational)
          (d                              . nat)
          (dlow                           . nat)
          (dhigh                          . nat)
          (dlazy                          . nat)
          (hbmInterval                    . pos-rational)
          (fanoutTTL                      . pos-rational)
          (mcacheLen                      . pos)
          (mcacheGsp                      . non-neg-rational)
          (seenTTL                        . non-neg-rational)
          (opportunisticGraftThreshold    . non-neg-rational)
          (topicWeight                    . non-neg-rational)
          (meshMessageDeliveriesDecay     . frac)
          (firstMessageDeliveriesDecay    . frac)
          (behaviourPenaltyDecay          . frac)
          (meshFailurePenaltyDecay        . frac)
          (invalidMessageDeliveriesDecay  . frac)
          (decayToZero                    . frac)
          (decayInterval                  . pos-rational)))

(defdata wp (cons weights params))
(defdata twp (map topic wp))
```

Note that Gossipsub required weights `w3` and `w3b` to be negative. However, the use of Defdata allowed us to automatically find that FileCoin used an invalid `twp` because it set `w3` and `w3b` to zero. We discussed this with the Gossipsub developers who then agreed to allow zero values. Given `T`, a set of topics which the neighbor subscribes to; `tctrs`, the neighbor's topic specific counters; `gctrs`, the neighbor's global counters; and a `twp` containing entries for each topic our neighbor subscribes to, the score function calculates a neighbor's score as shown below.

$$score(q) = \min(\mathrm{TC}, \sum_{\tau \in T} \mathrm{tw}^{\tau} \times \sum_{i \in \{1,2,3,3b,4\}} w_i^{\tau} P_i^{\tau}) + w_5 P_5 + w_6 P_6 + w_7 P_7$$

where

`(weights .  params)` = `(mget` τ `twp)`

$$
\begin{aligned}
w_i^\tau &= \texttt{(weights-w}\textit{i}\texttt{ weights)} \\
P_1^\tau &= \texttt{(calcP1(tctrs-meshTime tctrs) (params-meshTimeQuantum params)} \\
&\quad \texttt{(params-timeQuantaInMeshCap params))} \\
P_2^\tau &= \texttt{(calcP2(tctrs-firstMessageDeliveries tctrs) (params-p2cap params))} \\
P_3^\tau &= \texttt{(calcP3(tctrs-meshTime tctrs) (params-activationWindow params)} \\
&\quad \texttt{(tctrs-meshMessageDeliveries tctrs)} \\
&\quad \texttt{(params-meshMessageDeliveriesCap params)} \\
&\quad \texttt{(params-meshMessageDeliveriesThreshold params))} \\
P_{3b}^\tau &= \texttt{(calcP3b(tctrs-meshTime tctrs) (params-activationWindow params)} \\
&\quad \texttt{(tctrs-meshFailurePenalty tctrs)} \\
&\quad \texttt{(tctrs-meshMessageDeliveries tctrs)} \\
&\quad \texttt{(params-meshMessageDeliveriesCap params)} \\
&\quad \texttt{(params-meshMessageDeliveriesThreshold params))} \\
P_4^\tau &= \texttt{(calcP4(tctrs-invalidMessageDeliveries tctrs))} \\
P_5 &= \texttt{(gctrs-apco gctrs)} \\
P_6 &= \texttt{(gctrs-ipco gctrs)} \\
P_7 &= \texttt{(calcP7 (gctrs-bhvo gctrs))} \\
\mathrm{tw}^\tau &= \texttt{(params-topicweight params)} \\
\mathrm{TC} &= \texttt{(params-topiccap (cddar twp))}
\end{aligned}
$$

TC does not depend on any topic, but since it is stored in a `twp` which is indexed by `topic`, its value is replicated in each of the corresponding `params`. So, it is fine to extract its value from the first entry of a `twp`. Note that for score calculations, we require a non-empty `twp`.

Each of the `calcPi` functions where `i` $\in$ {`1,2,3,3b,4,7`} is used to calculate contributions to the score by one or more of counter values from `tctrs`. `calcP1` calculates the contribution to a neighbor's score based on the time spent in common meshes. `calcP2` awards score for being one of the first few to forward a message. `calcP3` calculates penalties due to the mesh message transmission rate being below a given threshold of `(params-meshMessageDeliveriesThreshold params)`. Whenever a peer is pruned, its corresponding `tctrs` counter `meshFailurePenalty` is augmented by the mesh message transmission rate deficit. This counter is not cleared even after the peer has been pruned. `calcP3b` scores mesh message delivery failures based on the value of this counter. Hence, P_{3b}^τ is a "sticky" value which is supposed to discourage a peer that was pruned because of underdelivery from quickly getting re-grafted in a mesh. `calcP4` calculates the penalty on score due to sending invalid messages. `calcP7` calculates penalties due to several kinds of misbehaviors described by the Gossipsub specification. An example of such misbehaviors includes spamming with too many `IHAVE` messages which are either bogus and/or not following up to the corresponding `IWANT` requests.

The Transition Function. We define a transition function `run-network`, which, given an initial `Group` state and a list of `evnt`, produces a trace of type `egl`, which is an alist of `evnt` and `group`. Hence, after running a simulation, we have access to the state of the

`Group` after each `evnt` was processed. `run-network` depends on the transition function for the `peer-state` (`transition`), which depends on the transition functions for `nbr-topic-state` (`update-nbr-topic-state`) and for `msgs-state` (`update-msgs-state`). For brevity, we mention only the signatures of each of these functions below. Notice that each of these signatures represents neatly and concisely the types of the formal arguments and the function return type, which is very useful in a large code base.

```
(defdata egl (alistof evnt group)) ;; simulation trace

(definecd run-network (gr :group evnts :loev i :nat r :twp s :nat) :egl
...)

(defdata peer-state-ret
  (record (pst . peer-state)
          (evs . loev)))

(definecd transition
  (self :peer pstate :peer-state evnt :evnt r :twp s :nat) :peer-state-ret
...)

(defdata msgs-state-ret
  (record (mst . msgs-state)
          (evs . loev)
          (tcm . pt-tctrs-map)
          (gcm . p-gctrs-map)))

(definecd update-msgs-state (mst :msgs-state evnt :evnt pcm :pt-tctrs-map
                             gcm :p-gctrs-map r :twp) :msgs-state-ret
...)

(defdata nbr-topic-state-ret
  (record (nts . nbr-topic-state)
          (evs . loev)
          (tcm . pt-tctrs-map)
          (gcm . p-gctrs-map)
          (sc  . peer-rational-map)))

(definecd update-nbr-topic-state (nts :nbr-topic-state
                                  nbr-scores :peer-rational-map
                                  tcm :pt-tctrs-map gcm :p-gctrs-map
                                  evnt :evnt r :twp s :nat) :
                                     nbr-topic-state-ret
...)
```

A Gossipsub peer can select a random subset of its fanout and promote them as mesh members. It can also selects a random subset of its neighbors to advertise with "IHAVE" messages. Such non-determinism is handled by sending a random seed `s` as a formal parameter to `run-network`, which is then propagated to the other transition functions it depends on.

Observe that a single event like full message forwarding can trigger several more forwards, causing a cascade of events. Such events are represented by the `evs` field in the return types of `update-msgs-state` and `update-nbr-topic-state`. In order to limit the total number of events processed, we send a natural number `i` as a formal parameter to the `run-network` function.

When proving contract theorems for the transition functions, we needed to prove the types of terms returned by utility functions, like `shuffle`, or ACL2 functions like `set-difference-equal`. For this, we used polymorphism and automated type-based reasoning provided by Defdata, as shown below:

```
(sig set-difference-equal ((listof :a) (listof :a)) => (listof :a))
(sig shuffle ((listof :a) nat) => (listof :a))
```

We made heavy use of higher-order macros written by Manolios [31] to improve the readability of our code. For example, `create-map*` is a list functor. Given an admitted function name or a lambda expression `f` of type `a` $\rightarrow$ `b`, `create-map*` defines a function `map*-*f` of type `(listof a)` $\rightarrow$ `(listof b)`. In the following code snippet, we show theorems proving that it obeys the functor laws, and give an example of its usage. `map*` is a syntactic sugar that maps function `f` onto a list without referring to the generated function name `map*-*f`.

```
(definec id (x :all) :all
  x)
;; Proof that create -map* is a list functor
;; 1) functor mapping preserves the identity function
(property functor-id (xs :tl)
  (== (map* id xs) ;; id is an identity function for lists
      (id xs)))

;; 2) functor mapping preserves function composition. f and g are declared
;; using defstub. gof is defined as a composition of g and f
(property functor-comp (xs :tl)
  (== (map* gof xs)
      (map* g (map* f xs))))

;; function to create a list of SND GRAFT events from peer p to a list of peers
(create-map* (lambda (tp p) `(,p SND ,(cdr tp) GRAFT ,(car tp)))
             lotopicpeerp
             loevp
             (:name mk-grafts)
             (:fixed-vars ((peerp p))))

(check= (map* mk-grafts '((FM . A) (DS . B)) 'P)
        '((P SND A GRAFT FM) (P SND B GRAFT DS)))
```

Given an admitted function name or a lambda expression `f` of type `a` $\times$ `b` $\rightarrow$ `b`, the higher order function `create-reduce*` defines a function `reduce*-*f` which accepts a list of elements of type `a`, an initial accumulator value of type `b`, and returns a reduction of the list using `f`, from left to right.

```
;; function to extract all the subscribers (neighboring peers) from a
   topic-lop-map
(create-reduce* (lambda (tp-ps tmp) (app tmp (cdr tp-ps)))
                lopp
                topic-lop-mapp
                (:name subscribers))

(check= (reduce* subscribers '()
                 '((T1 P1 P2 P3)
                   (T2 P4 P5 P1)))
        '(P4 P5 P1 P1 P2 P3))
```

6.2 Reasoning about the scoring function

Based on the observation that honest peers can be distinguished from malicious ones based on their observable behaviors (using local counters and scores), and thus, the overall network can be made more secure and performant if every honest peer promotes their well-behaving neighbors and demotes poorly-behaved ones, we came up with the following informal fundamental property.

Fundamental Property of Gossipsub Defense Mechanisms. *Peers who behave poorly will be demoted by their neighbors. Peers who behave better-than-average will be promoted by their neighbors. Promotion/demotion is entirely based on local peer behavior.*

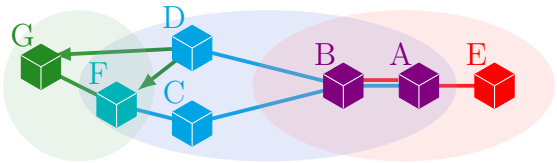


Figure 6.2.1: An example network

Before reasoning about the formalization of this fundamental property later in the chapter, we discuss the importance of this fundamental property. Consider the simple example shown in Figure 6.2.1, where peers A and B subscribe to, and are mesh neighbors in both Red and Blue topics. It might be possible for B to observe that A behaves perfectly well in the Blue topic while simultaneously misbehaving in the Red topic. This is not good for B because it depends on A for all of its messages in the Red topic. Note that this is a very simplified example. In an actual network, B could have several other neighbors subscribed to the Red topic. But as we will show later, it is equally trivial to have a scenario where all of B's neighbors isolate it from communications in the Red topic. In this example, we want B to prune A from its Red mesh in hopes of finding a better mesh neighbor later on. Reasoning about this fundamental property directly would be difficult due to the massive search-space of possible attack vectors. Hence, we focus on the following liveness property capturing the essence of the fundamental property:

Property 6.2.1. If a peer's score relating to its performance in any topic is continuously non-positive, then the peer's overall score should eventually be non-positive:

$$\forall q, \tau :: \langle \boldsymbol{G}(score(q) \textit{ for topic } \tau \leq 0) \Rightarrow \boldsymbol{F}(score(q) \leq 0) \rangle$$

where score(q) for topic t is defined below.

$$\text{tw}^{\tau} \times \sum_{i \in \{1,2,3,3b,4\}} w_i^{\tau} P_i^{\tau}$$

Notice that Property 6.2.1 is temporal. We write the non-temporal version of this property in context of Eth2.0 (using Eth2.0 `twp`) in ACL2s as shown below, and disprove the temporal version using an induction argument later in the dissertation.

Property 6.2.2.

```
(property (ptc :pt-tctrs-map pcm :p-gctrs-map p :peer top :topic)
  :hyps  (^ (member-equal '(,p . ,top) (acl2::alist-keys ptc))
            (> (lookup-score p (calc-nbr-scores-map ptc pcm *eth-twp*)) 0))
  (> (calcScoreTopic (lookup-tctrs p top ptc) (mget top *eth-twp*)) 0))
```

The following is one of the counter-examples to the above property, generated by cgen in ACL2s:

```
((top 'agg)
 (p 'p4)
 (pcm '((p3449 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))
        (p3450 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))
        (p3451 (:0tag . gctrs) (:apco . 0) (:bhvo . 0) (:ipco . 0))))
 (ptc '(((p4 . agg)
         (:0tag . tctrs)
         (:firstmessagedeliveries . 0)
         (:invalidmessagedeliveries . 0)
         (:meshfailurepenalty . 0)
         (:meshmessagedeliveries . 1)
         (:meshtime . 42))
        ((p4 . blocks)
         (:0tag . tctrs)
         (:firstmessagedeliveries . 324)
         (:invalidmessagedeliveries . 0)
         (:meshfailurepenalty . 0)
         (:meshmessagedeliveries . 330)
         (:meshtime . 377))
        ((p4 . sub1)
         (:0tag . tctrs)
         (:firstmessagedeliveries . 371)
         (:invalidmessagedeliveries . 0)
         (:meshfailurepenalty . 0)
```

```
         (:meshmessagedeliveries . 377)
         (:meshtime . 324))
        ((p4 . sub2)
         (:0tag . tctrs)
         (:firstmessagedeliveries . 318)
         (:invalidmessagedeliveries . 0)
         (:meshfailurepenalty . 0)
         (:meshmessagedeliveries . 324)
         (:meshtime . 371))
         ... ))
```

For brevity, we omit entries for peer-topic key values for peers other than `p4`. In property-based testing, the free variables of a property under test are assigned values using a synergistic combination of theorem proving and random assignments computed using type-based enumerators (generators) in an effort to discover counterexamples to the property. Observe that Property 6.2.2 depends on `ptc` and `pcm` which do not have trivial types. These are maps containing records which themselves consist of several numerical values, which makes the search space of possible counter-examples immensely large. In order to make it easier for cgen to find counter-examples, we wrote custom enumerators to enumerate restricted values for `topic`, `tctrs`, `gctrs`, `pt-tctrs-map` and `p-gctrs-map`. Specifically, we limit the penalties on the score due to some counters, such that the negative contributions due to misbehavior are comparable to the positive contributions due to good behavior. Below we define custom enumerators for `topic` and `tctrs`.

```
(definec topics () :tl
  ;; valid topics used in Ethereum
  '(AGG BLOCKS SUB1 SUB2 SUB3))

(definec nth-topic-custom (n :nat) :symbol
  (nth (mod n (len (topics))) (topics)))
(defdata lows (range integer (0 <= _ <= 1))) ;; high values
(defdata-subtype lows nat)
(defdata highs (range integer (300 < _ <= 400))) ;; low values
(defdata-subtype highs nat)

(defun nth-bad-counters-custom (n)
;; setting invalidMessageDeliveries and meshFailurePenalty to 0 due to high
   penalty
(defun nth-good-counters-custom (n)
  (tctrs 0 (nth-highs (+ n 2)) (nth-highs (+ n 3)) (nth-highs (+ n 4)) 0))

(defun nth-counters-custom (n) ;; custom enumerator for tctrs
  (if (== 0 (mod n 4))
      (nth-bad-counters-custom n)
    (nth-good-counters-custom n)))
```

Besides Property 6.2.2, we formalize three safety properties for GossipSub, stated below. Together, these four are the most general properties of the score function, which must hold in order for the fundamental property to hold.

Property 6.2.3. Increasing bad-performance counters (which are multiplied with negative weights) should decrease the overall score.

Property 6.2.4. Increasing good-performance counters (which are multiplied with positive weights) will not decrease the score for a mesh peer that has been in the mesh for a sufficiently long time.

Property 6.2.5. If two peers subscribe to the same topics, and achieve identical per-topic counters, and identical global counters, then they achieve identical scores.

We are able to find counterexamples to Property 6.2.3 in much the same way as 6.2.2, as shown in Table 6.1.

Topic	MT	FMD	FMD'	MMD	MMD'	IMD	MFP	Score	Score'
`Blocks`	147	194	3	200	10	0	0	22.21	6.21
`Agg`	150	194	194	230	230	0	0	13.83	13.83
`Sub1`	141	188	188	194	194	0	0	7.80	7.80
`Sub2`	110	180	180	232	232	0	0	7.80	7.80
`Sub3`	135	182	182	188	188	0	0	7.78	7.78
`Total`								32.72	32.72

Table 6.1: Perturbations in `topic-counters` values for an Eth2.0 peer that violate Property 2. Abbreviations denote score components: *FMD* (FirstMessageDeliveries), *FMD′* (FirstMessageDeliveries penalty), *MMD* (MeshMessageDeliveries), *MMD′* (MeshMessageDeliveries penalty), *IMD* (InvalidMessageDeliveries), *MFP* (MeshFailurePenalty), and *MT* (MeshTime). Both totals = 32.72, even though the count of FirstMessageDeliveries and MeshMessageDeliveries drops.

We manually prove that Property 6.2.4 holds over all configurations (available with the artifacts). The proof that Property 6.2.5 holds over all configurations follows directly from referential transparency of ACL2s.

6.3 Discussion: Relation to Floodsub and Meshsub Correctness Arguments

Property 6.2.2 formalizes a fundamental consistency expectation between local topic behavior and global peer reputation in Gossipsub v1.1. Informally, it states that a peer assigned a strictly positive neighbor score should also have strictly positive topic-specific score contributions. In other words, a peer's global reputation should not be supported by topic contexts in which its behavior is locally penalized.

This expectation aligns with the correctness intuitions underlying Floodsub and Meshsub. Although those protocols do not maintain explicit peer scoring, their refinement-based proofs establish that peers which publish and forward messages according to the protocol rules

preserve the safety and liveness properties of the dissemination overlay. Implicitly, these arguments assume behavioral coherence: peers that contribute positively to dissemination do not simultaneously violate topic-level assumptions.

From this perspective, Property 6.2.2 expresses a compatibility requirement between Gossipsub's scoring layer and the Commitment⇒Delivery property already established for Floodsub and Meshsub. The scoring mechanism is intended to ensure that peers exhibiting persistently poor topic-specific behavior eventually receive a non-positive overall score, thereby limiting their influence on mesh formation and message propagation.

Our counterexamples demonstrate that this property does not hold in Gossipsub v1.1. Peers with persistently non-positive topic scores may retain positive overall standing and remain in the dissemination mesh. This breakdown in the scoring mechanism directly undermines the Commitment⇒Delivery guarantee: there exist executions which lead to peer states, we found in our counter-examples, in which a message is committed at its origin, yet some peers subscribed at commit time never receive it. Thus, the failure of the score property provides a structural explanation for the violation of dissemination correctness. In the next section, we will synthesize such executions, as well as attacks that exploit vulnerabilities exposed by the failure of Gossipsub's scoring system.

Part III

Effectiveness of the Combined Methodology

Chapter 7

Model based Security Analysis of Gossipsub v1.1

Protocols intended for deployment in large networks must provide not only scalability and performance, but also strong resilience against adversarial behavior. As such systems grow in complexity, informal reasoning and testing alone are insufficient to establish confidence in their security, motivating the need for systematic verification techniques that can reason about all possible protocol executions. This need is particularly acute for Gossipsub, which was explicitly developed as a security-enhanced pubsub protocol designed to mitigate a wide range of known attacks.

The Gossipsub developers specified their protocol in English prose and reference implementations [47, 72, 81], and validated its security primarily through unit testing and network emulation of pre-programmed scenarios [50, 83]. These evaluations are intended to demonstrate that misbehaving peers in a Gossipsub network are eventually identified and pruned. However, as Dijkstra famously observed, "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." This motivated us to develop our own model of Gossipsub in ACL2s (as described in the previous chapter) that allows us to write and reason about properties about the protocol and its components, as well as to run simulations with precise control over network dynamics and states.

We synthesize and validate a collection of attacks on actual Eth2.0 topologies. Our attacks range from simple attacks with a single attacker and victim, to eclipse attacks, to attacks that partition Eth2.0 networks. All of the attacks are based on our attack gadgets, which encapsulate the essence of the vulnerabilities we discovered. Our attack gadgets can be applied to any network topology and allow an attacker to block or throttle certain message transmissions, without being discovered and without incurring any penalties. In fact, we show that for all of these attacks the scores assigned to the attackers by the victims stabilize; hence, by induction, they remain positive forever.

The counter-example 6.2 which we obtained in the previous section is a specification for an unsafe state that does not satisfy Property 6.2.2 for an Eth2.0 network. However, we are also interested in characterizing and generating attacks against an Eth2.0 Gossipsub network for three main reasons: (1) to show that an unsafe state is **reachable** from a reasonable start state, (2) **invalidation** of our temporal Property 6.2.1 using the trace generated from

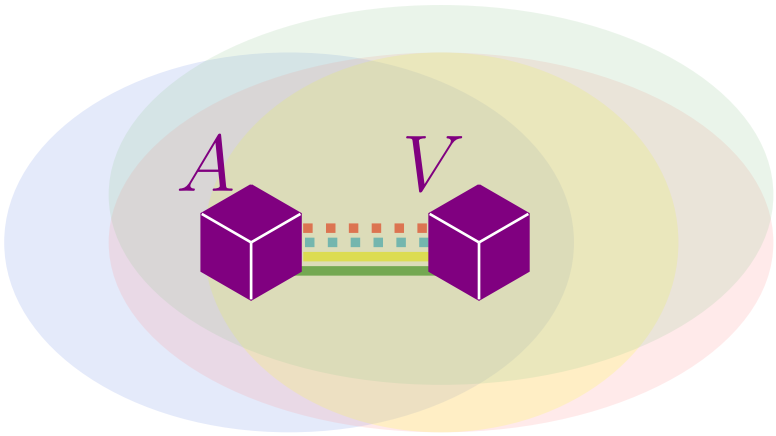


Figure 7.1.1: Example of an AG_2 attack gadget, where out of four common mesh topics, A is attacking V in the red and blue (dashed line) topics.

the attack, and finally (3) demonstration of **scalability** of our attack to large networks, typically of the size and shape used by real world applications.

The rest of this chapter is organized as follows. Section 7.1 defines attack gadgets and shows how to use them to create havoc in Gossipsub v1.1 networks. Section 7.2 describes real-world Eth2.0topologies used in our attack simulation. Section 7.3 details the experimental setup, while Section 7.4 interprets these attacks as refinement counterexamples—valid execution sequences that exploit Gossipsub v1.1's scoring mechanism to reach states unreachable in Floodsub and Meshsub.

Chapter provenance. This chapter is based on work previously published in [40, 41]. The presentation has been revised and extended for inclusion in this dissertation.

7.1 Attack Gadgets

An attack gadget is a tuple $\langle A, V, S\rangle$, where A, V are peers (A is the attacker and V is the victim), S is a set of subnet topics (the attacked topics), and A, V are mesh neighbors over a set of topics that is a superset of S. For each $i \in \mathbb{N}$, we define AG_i to be the set of attack gadgets where $|S| = i$. Figure 7.1.1 illustrates an example attack gadget in AG_2. The attack gadget allows A to maintain an overall positive score while misbehaving with respect to S, by behaving honestly with respect to the other topics. Therefore, if T is the set of subnet topics in the given Eth2.0 network, an attack gadget $\langle A, V, S\rangle$ in AG_i is only possible if $|T \setminus S|$ is large enough. Using the Eth2.0 GS parameters, we can calculate the number of other topics A and V have to subscribe to as follows.

$$\min\{t \in \mathbb{N} \mid (7.2 + 3.2\frac{t}{T} > 24.7\frac{i}{T}) \wedge (t + i \leq T)\} \tag{7.1}$$

The above only has a solution for some values of T. Also, note that to derive a corresponding formula for a different GS application, one has to use that application's parameters.

7.2 Eth2.0 Topologies

We validate our attacks using the full-network topologies of the Eth2.0 testnets Ropsten, Goerli and Rinkeby, as measured by Li *et al* [45] Table 7.1 shows basic characteristics of these Eth2.0 network topologies.

Network	Nodes	Degree			Diameter
		min	max	avg	
Ropsten	588	1	418	25.49	5
Goerli	1355	1	712	28.26	5
Rinkeby	446	1	191	68.96	6

Table 7.1: Eth2.0 Network Characteristics

Figure 7.2.1 shows a visualization of the Ropsten network topology, where each node is represented by a circle whose diameter is proportional to its degree. Similarly colored nodes are in the same community *i.e.*, share more edges among themselves than with the rest of the network.

7.3 Experimental Setup

Our experiments use the above topologies as follows. Given a topology and the number of subnet topics, T, we construct a corresponding ACL2S model (of type `Group`), with topics `BLOCKS`, `AGG` and T subnet topics, for $n = T + 2$ topics in total. Furthermore, every peer is a mesh member of every topic. We then generate attacks, as outlined below, and use ACL2S to check that the attacks are successful, *i.e.*, that the attackers *continuously* limit messages of the targeted topics without *ever* being discovered or penalized by any of the victims.

Single attacker/victim attacks We synthesize attacks involving a single attacker and a single victim by instantiating a single attack gadget. For each of the network topologies and for $i \in \{1, 2, 3\}$, where i corresponds to the number of attacked topics, we generate a corresponding ACL2S model. In the model, we determine the value of T, the number of subnet topics, using Equation 7.1. Once we have the ACL2S model, we generate a sequence of events consisting of message transmission events from A to V as well as heartbeat events at V (heartbeats are when V updates the scores of its peers). The shape of the events we generate is described by the following regular expressions:

$$\begin{aligned} \mathit{Msgs} &:= M_1^b \ldots M_i^b \; M_{i+1}^f \; \ldots M_n^f \\ \mathit{Events} &:= (\mathit{Msgs} \; H)^+ \end{aligned}$$

where (1) M_k^l denotes sending and receiving l payload messages from A to V for topic k, (2) $b \in \{0, 1\}$, as discussed below, (3) i is the number of attacked topics, as mentioned above,

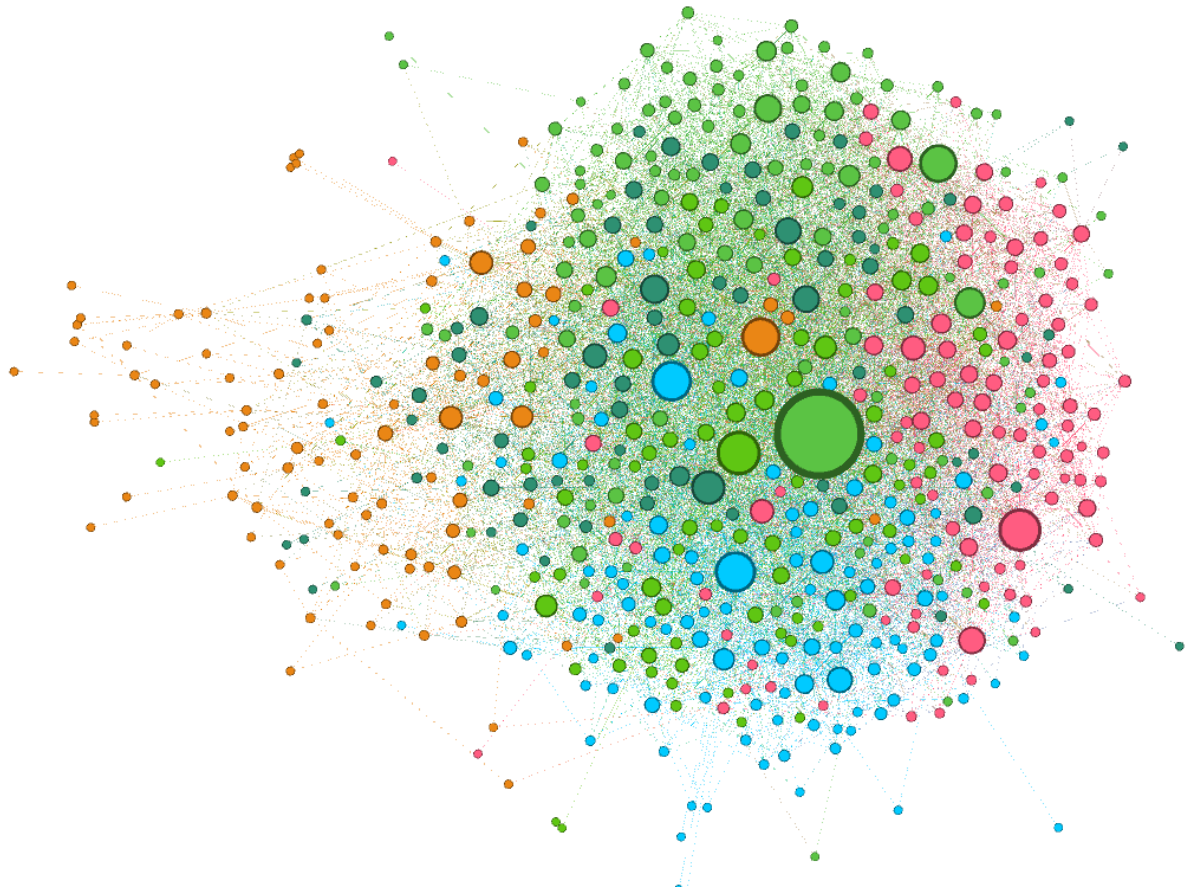


Figure 7.2.1: Visualization of the Ropsten topology of Eth2.0 nodes illustrating node degree distribution and community structure.

(4) f is the number of messages sent for the topics which are not attacked, as justified below, (5) $n = T + 2$ is the total number of topics, as described above, and (6) H is a heartbeat event at V.

The ordering of events is not important because any permutation of *Msgs* occurring between V's heartbeats will have the same effect on the network state. Also, we set f=10 for each topic because according to Eth2.0 *GS* parameters, under normal operation, this is at least 10% of the expected mesh message deliveries per topic and sending more than f messages can never decrease the score assigned to an attacker by the victim.

We synthesize two types of attacks. The first are *throttling attacks*, which reduce the mesh message transmission rate in the attacked topics to be below the threshold set by the Eth2.0 parameters by setting b=1. The second are *blocking attacks*, which completely block mesh message transmission for all of the attacked topics by setting b=0. We validate the attacks by checking that Property 1 is eventually always violated by the output trace of our experiments.

To test our model's performance on actual topologies, we ran our experiments on 100,000 events. Note that processing one event typically generates a cascade of other events because when V receives a message, it forwards it to its peers, who then forward it to their peers and

so on. Table 7.2 shows the time required to simulate and validate our attacks in ACL2s, on a 16GB M1 Macbook Air machine.

Network	Throttling	Blocking		
	AG_1	AG_1	AG_2	AG_3
Ropsten	1.3	1.3	1.3	1.3
Goerli	1.3	1.4	1.8	1.1
Rinkeby	1.6	1.9	2.0	2.1

Table 7.2: Wallclock time (mins) for attack simulation

We observed that for all experiments, the first violation of Property 1 is observed at event H occurring right after the activation period (an Eth2.0 parameter) has passed. Hence, an attacker peer can start its attack quickly after it joins a mesh. Experimentally, we observed that this attack is not transient as attack scores (assigned by victims) eventually converge to a positive number that stays the same in successive heartbeats at V. By induction, this establishes that our attacks are perpetual. The time taken by our models to process 100,000 events gives a sense of the performance one can expect for other kinds of simulations. Simulation times for the remaining attacks are similar and for brevity, we only simulate these attacks until stability which is achieved, which never takes more than 5 seconds.

7.3.1 Eclipse Attacks

We can use attack gadgets to construct other eclipse attacks by just instantiating an attack gadget per neighbor of a victim such that if they collude, they can target and completely isolate the victim *i.e.*, the victim will never receive any messages in the i attacked topics. We tested this attack in the Ropsten topology by identifying a victim node with four neighboring peers (about 12% nodes have degree less than 5; see Figure 7.2.1) and instantiating its neighbors as attackers using AG_3 gadgets. We verified that the victim's message cache contained no message received in any of the attacked topics while containing messages received in non-attacked topics, that the attackers were continuously assigned positive scores by the victim, and that this behavior was perpetual.

7.3.2 Partition Attacks

Attack gadgets can also be used to isolate a subset of the peers in a network by carrying out a network partition attack, which we describe as follows. Given a network graph $G = \langle V, E \rangle$ and a set of victims S, we want to identify a set X, preferably of minimal cardinality, such that X is a *vertex cut* of G that partitions G into disconnected components $\{S, V \setminus \{S \cup X\}\}$. The elements of X are the misbehaving peers, *i.e.*, each peer in X attacks all of its

non-X neighbors, using our attack gadgets to throttle or block messages over the attacked topics. Hence, no message in an attacked topic can be sent to a peer in S from a peer outside of the partition and vice versa. In addition to targeting specific victims, interesting partition attacks include setting S to be a community of the network (since partitions along community lines are expected to require smaller vertex cuts than are required for other sets of peers; see Figure 7.2.1) and choosing S such that $|S|$ is $\approx \frac{|V|}{2}$, which will bisect the graph. Finding minimal vertex cuts is an NP-hard problem [6], and can be reduced either to a Pseudo-Boolean problem or to 0-1 Integer Linear Programming problem. We synthesized and evaluated partition attacks using the Ropsten topology by selecting a set of victims, S, where $|S| = 6$, finding a minimal vertex cut, X (in our case $|X| = 2$) and creating the appropriate attack gadgets. We verified that each of the victim's message caches contained no message in any of the attacked topics that originated outside of S, but did contain messages in non-attacked topics received from outside of S, that victim nodes continuously assigned positive scores to their attackers (the nodes in X) and that this behavior was perpetual, *i.e.*, maintained forever.

7.4 Discussion: Attacks as Refinement Counterexamples

The attacks described above can be understood as concrete witnesses to a breakdown in behavioral assumptions that are implicitly relied upon by earlier publish—subscribe protocols. Although these attacks are constructed against Gossipsub v1.1, they exploit executions that would be unreachable under the correctness arguments developed for Floodsub and Meshsub.

Refinement-style reasoning for Floodsub and Meshsub establishes that well-behaving peers preserve the dissemination guarantees of the overlay under all admissible executions. Since these protocols do not maintain adaptive reputation or scoring state, they cannot reach configurations in which a peer is structurally favored while simultaneously degrading topic-level dissemination. As a result, the classes of executions exploited by our attacks are ruled out by construction.

Gossipsub v1.1, by contrast, introduces peer scoring as an additional layer of state that directly influences mesh maintenance decisions. This additional flexibility admits executions in which mesh membership becomes decoupled from topic-level behavior. The attacks exploit this decoupling, driving the protocol into states that violate the behavioral invariants implicitly preserved by Floodsub and Meshsub.

Viewed through this lens, the attacks function as refinement counterexamples. They arise not from malformed inputs or undefined behavior, but from sequences of locally valid protocol transitions. Property-based testing enables the systematic discovery of such executions without requiring the construction of an explicit refinement proof, providing a lightweight means of identifying refinement-level failures in security-augmented protocol designs.

Chapter 8

Lean Gossip–Simulation-Based Optimization of Gossipsub

The formal refinement analysis of `Meshsub` identifies the gossip fanout parameter D_{lazy} and the recently-seen window size W as the two correctness-critical parameters (conditions (C1) and (C2)). The refinement proof establishes that $D_{lazy} \geq 1$ ensures every heartbeat round advertises m to at least one non-mesh neighbor, and $W \geq 1$ ensures every forwarded message remains eligible for gossip for at least one round — together guaranteeing that *OverlayTConnectedp* is satisfiable. The proof proceeds via a commitment map `m2f` (Definition 5.3.28) that abstracts away in-transit messages from both the mesh forwarding phase and the gossip phase, producing a quiescent `Floodsub` state only when m has fully left the frontier $\mathcal{F}(s)$. The mesh degree D and fanout D_{lazy} influence dissemination latency and bandwidth in practice, but the correctness proof does not prescribe concrete values.

The Gossipsub parameters D and D_{lazy} were originally selected heuristically to ensure attack resilience [74, 75] and maintain "balance" between bandwidth and speed [75, 82], with limited quantitative justification. The official documentation states that "D_{lazy} is considered optional... By default, we simply use D for both" [48]. In this chapter, we complement our formal analysis with large-scale simulation to explore the quantitative impact of D and D_{lazy} on Gossipsub performance, using simulation-based optimization to identify parameter regimes that achieve favorable trade-offs between delivery rate, bandwidth cost, and latency.

This chapter introduces and evaluates what we call *Lean Gossip*: a configuration regime for gossip-based dissemination protocols in which recovery via gossip is deliberately kept minimal. Concretely, Lean Gossip refers to Gossipsub configurations with sparse eager meshes (D small) and a minimal gossip degree ($D_{lazy} = 1$), relying on gossip as a lightweight recovery mechanism rather than a primary dissemination channel. As we show, this minimal use of gossip is sufficient to compensate for churn-induced message loss while avoiding the bandwidth explosion associated with aggressive gossiping or redundant eager-push flooding.

While these parameters are known to influence typical performance, our results show that Gossipsub with minimal gossip (D_{lazy}=1) dominates Floodsub on delivery rate and bandwidth cost across all churn levels, achieving comparable reliability at 29–35× lower bandwidth cost. This provides the first principled guidance for parameter selection: sparse mesh connectivity with gossip-based recovery outperforms redundant eager-push connections, and Ethereum's deployed mesh degree lies near the Pareto frontier of delivery—cost

tradeoffs.

We make the following contributions:

First systematic parameter study of Gossipsub under churn. We present the first comprehensive simulation-based evaluation of Gossipsub parameterization across mesh degree D and gossip degree D_{lazy} under four churn regimes (0–30%) in a 1000-node network. Our simulator implements Gossipsub v1.0 semantics and is validated against published Protocol Labs measurements.

Floodsub lies off the delivery—bandwidth Pareto frontier under churn. We show that simple flooding (Floodsub) is dominated by Gossipsub in the delivery—bandwidth plane under non-zero churn (10–30%). For every such churn level, there exist Gossipsub configurations that achieve both higher delivery and lower bandwidth cost simultaneously. At 0% churn, Floodsub achieves 100% delivery which sparse Gossipsub meshes do not match, so the dominance relationship holds only under churn. Floodsub's only consistent advantage across all churn levels is lower latency.

Lean Gossip ($D_{lazy} \leq 3$) strictly dominates Ethereum's deployed configuration across all performance dimensions. We identify *Lean Gossip* configurations ($D_{lazy} \leq 3$) that strictly dominate Ethereum's deployed Gossipsub parameters across all three objectives simultaneously. At 20% churn, (D=8, D_{lazy}=3) achieves 99.9% delivery, 4.9 s p99 latency, and 34× bandwidth cost—beating Ethereum's 99.8% delivery and 106× cost while matching its latency, at 3.2× lower cost. The leaner (D=8, D_{lazy}=1) reduces cost to 9.8× (nearly 11× cheaper than Ethereum) while maintaining lower p99 latency and sacrificing only 0.5% points of delivery.

Recovery dominates redundancy. We demonstrate that gossip-based recovery—not eager-push redundancy—is the dominant mechanism for reliable dissemination under churn. Increasing D beyond approximately 8 yields diminishing delivery returns, while increasing D_{lazy} beyond 1 sharply increases bandwidth cost for negligible delivery improvement (0.3 percentage points across $D_{lazy} \in [1, 21]$ at 20% churn).

Pareto dominance framework for protocol evaluation. We introduce a two-dimensional Pareto dominance analysis over worst-case delivery and normalized bandwidth cost, enabling principled, multi-objective comparison of dissemination strategies.

The formal model developed in Chapter 5 establishes that Meshsub correctly implements Broadcastsub under the stated correctness conditions. However, that model intentionally abstracts away timing, network latency, bandwidth consumption, and realistic churn patterns. The simulation study in this chapter complements the formal analysis by quantifying these performance characteristics under realistic network conditions.

The rest of this chapter is organized as follows. Section 8.1 describes our simulation based methodology. Section 8.2 details our implementation of Gossipsub v1.0 used for simulation

analysis. Section 8.3 presents the simulation results, and Section 8.4 discusses their implications. Section 8.5 discusses limitations of our work and scope for future work. Section 8.6 discusses related work.

Chapter provenance. A version of this chapter is currently under review.

Use of Generative AI Tools We used generative AI tools, specifically Claude Opus 4.5 and ChatGPT 5.2, to accelerate the development of simulation code and for editing parts of this section. All AI-generated content was critically reviewed, revised, and validated manually. The simulation logic, experimental design, analysis, and interpretation of results remain entirely our work, and we take full responsibility for the accuracy and integrity of this section.

8.1 Methodology

This section describes the simulation framework used to evaluate Gossipsub dissemination as a function of mesh degree D and gossip degree D_{lazy} under churn. We isolate protocol-level delivery—bandwidth—latency tradeoffs by modeling a single-topic overlay and excluding scoring and multi-topic interactions.

8.1.1 Gossipsub Overview

Gossipsub combines eager push via a maintained mesh overlay with lazy pull via metadata gossip. Each peer maintains a neighbor set $p.nbrs$ and, for a subscribed topic t, an eager mesh $p.\mathcal{M}(t)$ of size approximately D to which full payloads are forwarded. Periodically, peers gossip message identifiers to up to D_{lazy} non-mesh neighbors, enabling recovery via IHAVE/IWANT exchanges. Heartbeat maintenance enforces mesh degree bounds and emits gossip.

For a subscribed topic $s \in p.\mathcal{S}$, peer p maintains a *mesh* $p.\mathcal{M}(s) \subseteq \{q \in p.nbrs : s \in q.\mathcal{S}\}$, a subset of s-subscribing neighbors to which p eagerly forwards full message payloads upon first receipt. The mesh degree is bounded: $D_{lo} \leq |p.\mathcal{M}(s)| \leq D_{hi}$, where D is the target degree and by default D_{lo} =D−2, D_{hi} =D+2. Likewise, for an unsubscribed topic $u \in p.\mathcal{U}$, peer p maintains a *fanout* $p.\mathcal{F}(u) \subseteq \{q \in p.nbrs : u \in q.\mathcal{S}\}$, enabling message origination on topics without subscription. Meshes, fanouts, and subscriptions are all mutable: a peer may unsubscribe from a topic, delete its corresponding mesh, and build a fanout; or vice versa.

Metadata dissemination. Each peer maintains a bounded message cache $p.mcache$ mapping recent message identifiers to payloads. Metadata about recently received messages are periodically broadcast to up to D_{lazy} randomly selected neighbors outside the mesh, called *gossip peers*. These IHAVE announcements contain message identifiers from $p.mcache$, allowing metadata to disseminate quickly with low overhead. Upon receiving IHAVE(M) from peer q, a peer p identifies missing messages $M' = M \setminus p.seen$ and responds with IWANT(M'), after which q transmits the requested payloads. This lazy pull mechanism enables recovery of messages missed due to mesh disconnection.

Heartbeat maintenance. Gossipsub executes a periodic heartbeat procedure at configurable intervals (default 1 s) to maintain mesh structure and disseminate metadata. For each topic $t \in p.\mathcal{S}$, the heartbeat performs the following operations in sequence:

Mesh maintenance. Enforce degree bounds by grafts and prunes.

$$|p.\mathcal{M}(t)| < D_{lo} \Rightarrow \text{select } k = D - |p.\mathcal{M}(t)| \text{ peers from } p.nbrs \setminus p.\mathcal{M}(t), \text{send } \mathsf{GRAFT} \text{ to each, add to } p.\mathcal{M}(t)$$
$$|p.\mathcal{M}(t)| > D_{hi} \Rightarrow \text{select } k = |p.\mathcal{M}(t)| - D \text{ peers from } p.\mathcal{M}(t), \text{send } \mathsf{PRUNE} \text{ to each, remove from } p.\mathcal{M}(t)$$

Fanout maintenance. For each topic $u \in p.\mathcal{U}$ with non-empty fanout, remove peers that have unsubscribed and replenish if $|p.\mathcal{F}(u)| < D$. Expire fanouts inactive beyond a configurable TTL.

Gossip emission. Select up to D_{lazy} peers from $p.nbrs \setminus p.\mathcal{M}(t)$ and emit IHAVE announcements containing message identifiers from $p.mcache$.

Cache shift. Advance the message cache window, expiring entries older than the gossip history length (default 3 heartbeat intervals). This bounds memory usage while retaining sufficient history for IWANT responses.

8.1.2 Model Simplifications

To isolate dissemination dynamics, we apply the following simplifications:

Single topic. All peers subscribe to one topic ($|\mathcal{T}| = 1$), eliminating fanout interactions and cross-topic effects.

No peer scoring. We omit v1.1 scoring, as it addresses adversarial behavior rather than benign churn.

Static neighbor sets. Peer tables are initialized once and change only due to churn.

No Floodsub compatibility. We model pure Gossipsub semantics.

Under these assumptions, peer state reduces to mesh neighbors, gossip peers, pending queue, seen set, bounded message cache, availability state, and delivery record. The parameters D and D_{lazy} fully determine redundancy.

Simplified peer state. Under these simplifications, peer state reduces to a septuple. For the single topic $t \in \mathcal{T}$, we write: $p.\mathcal{M} = p.\mathcal{M}(t)$: the eager mesh neighbors, $p.\mathcal{L} \subseteq p.nbrs \setminus p.\mathcal{M}$: the gossip peers selected for IHAVE emission, $p.pending$: the queue of inbound messages yet to be processed, $p.seen$: the set of received message identifiers (cleared on rejoin), $p.mcache$: the bounded message cache, $p.\omega \in \{\mathsf{online}, \mathsf{offline}\}$: the availability state, $p.delivered$: a persistent record of all messages received by peer p, which survives peer rejoin events.

The parameters D (target mesh degree) and D_{lazy} (maximum gossip degree) fully determine the protocol's redundancy—efficiency tradeoff in this simplified model. Note that $p.seen$ is used for duplicate detection during protocol execution and is cleared when a peer rejoins, while $p.delivered$ is used solely for delivery measurement and persists across churn events to ensure accurate delivery rate statistics.

These simplifications preserve the core dissemination dynamics: eager mesh forwarding provides primary delivery, while lazy pull serves as a recovery mechanism for messages missed

due to mesh disconnection. Our single-topic model essentially captures the behavior of either a mesh overlay (when peers subscribe) or a fanout overlay (when peers originate messages without subscription) in a full multi-topic Gossipsub network. Since both structures share identical eager-push semantics and gossip recovery mechanisms, our model isolates the fundamental dissemination dynamics common to all Gossipsub overlays.

We implement Gossipsub v1.0 semantics, omitting the peer scoring mechanism introduced in v1.1 [49]. Peer scoring is primarily for building attack resilience by penalizing misbehaving peers and is orthogonal to dissemination dynamics. This simplification isolates the core delivery—efficiency tradeoffs arising from mesh structure and gossip parameters.

8.2 Protocol Implementation

Having formally described the protocol in the previous section, we now explain key design decisions, evaluation metrics, and protocol parameter values used in our simulation.

8.2.1 Aggregate Message Workload

Table 8.1: Ethereum message sizes

Category	**Size (Bytes)**
Attestation	512
Aggregated attestation	1536
Beacon block / Blob	131072
Message identifier	32
IHAVE / IWANT	32 / 16
GRAFT / PRUNE	100

Ethereum's Gossipsub layer disseminates a heterogeneous mix of beacon blocks, attestations, aggregated attestations, and protocol control messages (Table 8.1). Beacon blocks (blobs) are large (approximately 128 KiB) but rare, with only one block produced per 12-second slot [19]. In contrast, attestation-related messages dominate by count. Empirical measurements of the Ethereum P2P network show that attestation-derived messages dominate gossip traffic by count, while beacon blocks are comparatively rare [24]. Individual attestations are aggregated prior to gossip dissemination, making aggregated attestations the primary message type processed by the Gossipsub layer. Accordingly, we model message dissemination using a single representative aggregated message size of 1.5 KiB which dominates Ethereum gossip traffic by volume (Table 8.1).

According to Ethereum's attestation workload, each peer publishes messages according to a Poisson process with rate $\lambda_{\text{pub}} = v/384$, where v denotes the number of validators colocated with the peer. We primarily evaluate the case $v = 1$, yielding roughly one message every 384 seconds per peer. We use a Poisson process to smooth slot-synchronous effects while preserving Ethereum's long-run attestation rate.

8.2.2 Network Model

Topology. We simulate a network of 1000 peers. We verified that qualitative trends remain unchanged for larger network sizes; we use 1000 peers to enable exhaustive parameter sweeps. Each peer maintains a neighboring peer table of 40–80 connections drawn from this population. The peer table size is consistent with measurements of Ethereum consensus nodes showing approximately 60 average connections per node [34]. Eager and lazy neighbors are sampled uniformly from this table.

Network Delay Models. We evaluate our simulator under two network delay models:

Constant delay model. All per-hop latencies are fixed at 25 ms, matching Protocol Labs' 50 ms RTT assumption [74]. We use this model for validation (Section 8.2.8).

Geographic delay model. Nodes are partitioned into three regions reflecting Ethereum's geographic distribution [71]: 30% Europe, 40% North America, and 30% Asia. Per-hop latencies are drawn from region-pair-specific normal distributions (Table 8.2). **We use this model for all parameter sweep experiments (Section 8.2.8)**, as it better reflects production Ethereum network conditions.

	Europe	**N. America**	**Asia**
Europe	$\mathcal{N}(30, 10^2)$	$\mathcal{N}(90, 20^2)$	$\mathcal{N}(180, 30^2)$
N. America	—	$\mathcal{N}(40, 10^2)$	$\mathcal{N}(150, 25^2)$
Asia	—	—	$\mathcal{N}(35, 10^2)$

Table 8.2: Per-hop latency distributions (ms) by region pair

The effective per-hop mean under this model is:

$$\mu_h^{\text{eff}} = \sum_{r,s} p_{rs} \cdot \mu_{rs} \tag{8.1}$$

where p_{rs} is the proportion of edges between regions r and s. With random peer selection, the effective per-hop mean latency is $\mu_h^{\text{eff}} \approx 105$ ms.

Churn. Based on real-world measurements by Kiffer *et al* [34], we model churn using a bimodal peer classification. A fraction c of peers are designated as "churny" and cycle between online and offline states with transition probabilities $\lambda_{\text{leave}} = 0.01$ and $\lambda_{\text{rejoin}} = 0.05$ per tick, resulting in rapid cycling with mean connection duration of 10 seconds, consistent with the short-lived connections observed in Ethereum's P2P network [34]. The remaining $1-c$ "stable" peers remain online throughout the experiment ($\lambda_{\text{leave}} = 0$). Note that when an offline peer comes back online, it loses all its state except its churn type and peer table (ambient peer discovery). At steady state, churny peers maintain approximately 83% availability, resulting in an overall network availability of $(1 - c) + 0.83c$. We evaluate configurations with $c \in \{0.0, 0.1, 0.2, 0.3\}$.

8.2.3 Simulation Timing

The simulator advances in discrete ticks of duration $\Delta t = 0.1$ seconds. Gossipsub heartbeat routines are executed every 10 ticks, corresponding to a 1 sec default heartbeat interval. At each tick the following events occur: (1) token-bucket refills for all peers, (2) message production according to a Poisson process with rate $\lambda_{\text{pub}} = 1/384$ per peer, (3) processing of scheduled message deliveries according to per-link delays, (4) churn transitions (online/offline state changes of churny peers), (5) bandwidth enforcement via send accounting, which drops messages exceeding per-peer bandwidth limits, and (6) execution of heartbeat routines (every 10 ticks) including IHAVE emission and mesh maintenance.

Message processing model. Messages arriving within the same tick are processed sequentially in arbitrary order. With $\Delta t = 0.1$ s and per-hop delays of 30-200 ms, simultaneous arrivals are rare; this discrete-time approximation preserves the invariant that each message is processed at most once per peer via the *seen* set.

Processing rate. We assume a conservative processing capacity of one payload and one control message per 100 ms tick (approximately 10 messages/sec). Higher service rates reduce absolute latency but do not change relative ordering across configurations.

We discard the first 2000 ticks (200 seconds) as warmup to allow the mesh to stabilize before measuring metrics. Algorithm 5 formalizes this tick-based execution model.

8.2.4 Bandwidth Model

Token-bucket approximation of TCP We model upload bandwidth using a token-bucket abstraction of TCP congestion control. Each peer has capacity B (25 Mbps), bucket size $B/8$ bytes, and refill rate $B \cdot \Delta t/8$ per tick. A message of size s is transmitted only if sufficient tokens are available. This permits short bursts while enforcing long-term throughput limits. Across all experiments, bandwidth exhaustion did not cause delivery loss; message loss arises solely from churn.

Implication for Floodsub comparison. This bandwidth setting is deliberately generous: 25 Mbps is sufficient to sustain Floodsub's 60-peer eager push without token-bucket drops. Under tighter bandwidth constraints—realistic for home validators on residential connections—Floodsub's high per-peer fanout would saturate the upload budget first, causing bandwidth-induced delivery loss that our experiments do not capture. Our results therefore represent a *conservative* estimate of Floodsub's disadvantage: in bandwidth-constrained deployments, the delivery gap between Floodsub and sparse Gossipsub configurations would widen further.

8.2.5 Evaluation Metrics

We evaluate Gossipsub configurations along three axes: delivery, efficiency, and latency. For each metric, we define both per-message and aggregate measures. Let V denote the set of all peers in the network. At any time t, a subset of peers are online and participating in the protocol; we denote this set as $V_t \subseteq V$.

Definition 8.2.30 (Delivery Set)

The *delivery set* $R_{m,t}$ is the set of peers that received message m by time t during the simulation:

$$R_{m,t} = \{p \in V : m \in p.\textit{delivered} \text{ by time } t\} \tag{8.2}$$

where *delivered* is a persistent record of all messages received by peer p, which survives peer rejoin events.

Definition 8.2.31 (Delivery rate)

We define the time-dependent delivery rate $\delta_{m,t}$ for $t \geq t_m$ as:

$$\delta_{m,t} = \frac{|R_{m,t}|}{|V|} \tag{8.3}$$

This captures the evolution of message dissemination through the network. At publication time, $\delta_{m,t_m} = 0$ since no peer has yet received the message—not even the publisher v_m, which inserts m into its *pending* queue before m is processed, forwarded to mesh peers, and added to $v_m.\textit{seen}$. As the protocol propagates m through eager-push and gossip-based recovery, $\delta_{m,t}$ increases monotonically, *i.e.*, $\langle \forall t_i, t_j :: t_i \leq t_j \Rightarrow \delta_{m,t_i} \leq \delta_{m,t_j} \rangle$. Let f_m be the first time after t_m when m is no longer in the *pending* queue of any peer and all gossip-based recovery attempts have completed. After f_m, m will not be forwarded again. In a stable network without churn, we expect $\delta_{m,f_m} = 1$. However, under churn, some peers in V may go offline before receiving m, causing δ_{m,f_m} to stabilize below 1. We define the *final delivery rate* $\delta_m = \delta_{m,f_m}$.

Aggregate Metrics

Definition 8.2.32 (Mean Delivery Rate)

For a simulation producing message set M, the mean delivery rate is:

$$\bar{\delta} = \frac{1}{|M|} \sum_{m \in M} \delta_m \tag{8.4}$$

Our simulation computes δ_m at the end of the measurement period T rather than waiting for stabilization at f_m. Messages published near the end of the measurement period may still have m in some peer's *pending* queue when statistics are computed, yielding $\delta_{m,T} \leq \delta_m$. This slightly underestimates delivery rates, particularly for configurations with high tail latency. However, since this effect applies uniformly across all configurations, relative comparisons remain valid.

Definition 8.2.33 (Total Bandwidth)
Total bytes transmitted during the measurement period:

$$B_{total} = \sum_{p \in V} \left(B_p^{\texttt{Payload}} + B_p^{\texttt{Ctrl}} \right) \tag{8.5}$$

where $B_p^{\texttt{Payload}}$ counts message payload bytes and $B_p^{\texttt{Ctrl}}$ counts control message (`IHAVE`, `IWANT`, `GRAFT`, `PRUNE`) bytes sent from peer p.

Definition 8.2.34 (Bytes Per Delivery)
Quantifies bandwidth efficiency by measuring the average bytes transmitted per successful delivery:

$$\beta = \frac{B_{total}}{\sum_{m \in M} |R_m|} \tag{8.6}$$

This metric captures protocol overhead from multiple sources: redundant payload transmissions (*e.g.*, eager push to multiple mesh neighbors), control messages for lazy push or mesh maintenance, and wasted bandwidth from transmissions to peers that subsequently depart. Lower values indicate better resource utilization, with the theoretical minimum corresponding to perfect amortization of a single payload transmission across all recipients.

Definition 8.2.35 (Normalized Cost)
We normalize by payload size $P = 1536$ bytes:

$$\beta / P \tag{8.7}$$

This normalization yields a dimensionless measure capturing the average multiple of a payload's worth of traffic required per successful delivery, enabling direct comparison across parameter settings independent of payload size. A normalized cost of 10 indicates that, on average, 10 payload-equivalents of total traffic are transmitted per successful delivery.

Definition 8.2.36 (Per-Delivery Latency)
For each successful delivery (m, p) where $p \in R_m$, the latency is:

$$\ell_{m,p} = (t_{m,p} - t_m) \cdot \Delta t \tag{8.8}$$

where $t_{m,p}$ is the tick at which peer p first received message m in its pending queue.

Definition 8.2.37 (P99 Latency)

The 99th percentile latency is computed over all successful deliveries:

$$p99 = Percentile_{99}\left(\{\ell_{m,p} : m \in M, p \in R_m\}\right) \tag{8.9}$$

High $p99$ may indicate suboptimal mesh connectivity, high churn, or protocol mechanisms (such as lazy pull) that delay message propagation under certain conditions. Latency is used only as a relative ordering metric for Pareto dominance; we do not claim that absolute $p99$ values predict production behavior.

Together, these metrics characterize the delivery-efficiency-latency tradeoff space that distinguishes gossip protocol designs. An effective protocol must balance high delivery rates against bandwidth costs while maintaining acceptable latency bounds. All results are averaged across three independent random seeds to ensure statistical confidence.

Algorithm 5 Gossipsub Simulation Main Loop

Require: Network parameters (n, D, D_{lazy}, c), simulation length T, warmup T_w
Ensure: Metrics $(\bar{\delta}, \beta_{norm}, p99)$

Initialize n peers with peer tables of size $[40, 80]$
Initialize eager mesh $\mathcal{M}_p$ and gossip peers $\mathcal{L}_p$ for each peer p
$t \leftarrow 0$
while $t < T_w + T$ **do**
    RefillTokenBuckets ▷ Bandwidth accounting
    ProduceMessages(t) ▷ Poisson arrivals
    ProcessDeliveries(t) ▷ Scheduled network arrivals
    ProcessChurn(t) ▷ Bimodal leave/rejoin
    EnforceBandwidth(t) ▷ Drop excess messages
    if $t \mod 10 = 0$ **then** ▷ Heartbeat every 1 second
        EmitGossip(t) ▷ IHAVE announcements
        MaintainMesh(t) ▷ GRAFT/PRUNE
    end if
    $t \leftarrow t + 1$
    if $t = T_w$ **then**
        Reset all counters ▷ End of warmup
    end if
end while
return ComputeMetrics

Algorithm 6 Token Bucket Refill

procedure RefillTokenBuckets
    for all peer p **do**
        $p.tokens \leftarrow \min\big(B/8,\ p.tokens + B \cdot \Delta t/8\big)$
    end for
end procedure

Algorithm 7 Message Production

procedure ProduceMessages(t)
  for all online peer p **do**
    $\lambda \leftarrow v_p/384$ ▷ v_p validators, one attestation per epoch
    $p_{pub} \leftarrow 1 - e^{-\lambda \cdot \Delta t}$ ▷ Poisson probability
    if $Uniform(0,1) < p_{pub}$ **then**
      $m \leftarrow$ new message with $origin = p$, $t_m = t$
      $p.seen \leftarrow p.seen \cup \{m\}$
      $p.mcache \leftarrow p.mcache \cup \{m\}$
      $p.delivered \leftarrow p.delivered \cup \{m\}$
      for all $q \in p.\mathcal{M}$ **do** ▷ Eager forward to mesh
        if $\omega_q = \mathsf{online}$ **then**
          Schedule delivery of m from p to q at tick $t + \text{Delay}(p, q)$
        end if
      end for
    end if
  end for
end procedure

Algorithm 8 Process Scheduled Deliveries

procedure ProcessDeliveries(t)
  for all $(m, sender)$ scheduled for delivery to peer p at tick t **do**
    if $\omega_p = \mathsf{offline}$ **then**
      continue ▷ Drop: peer offline
    end if
    if $m \in p.seen$ **then**
      continue ▷ Duplicate: already processed
    end if
    $p.seen \leftarrow p.seen \cup \{m\}$
    $p.mcache \leftarrow p.mcache \cup \{m\}$
    $p.delivered \leftarrow p.delivered \cup \{m\}$ ▷ Persistent record
    Record $t_{m,p} \leftarrow t$ ▷ For latency measurement
    for all $q \in p.\mathcal{M} \setminus \{sender\}$ **do** ▷ Eager forward to mesh
      if $\omega_q = \mathsf{online}$ **then**
        Schedule delivery of m from p to q at tick $t + \text{Delay}(p, q)$
      end if
    end for
  end for
end procedure

Algorithm 9 Network Delay Model

function DELAY(p, q)
  $r_p, r_q \leftarrow$ regions of peers p and q
  $(\mu, \sigma) \leftarrow \mathit{LatencyTable}[r_p, r_q]$ ▷ See Table 8.2
  $d \leftarrow \max(0.01, \mathcal{N}(\mu, \sigma^2))$ ▷ Sample, clip to 10ms minimum
  return $\lceil d / \Delta t \rceil$ ▷ Convert seconds to ticks
end function

Algorithm 10 Bimodal Churn Model

procedure PROCESSCHURN(t)
  for all peer p **do**
    if $p.\mathit{type} =$ churny **then**
      $\lambda_{\mathit{leave}} \leftarrow 0.01, \quad \lambda_{\mathit{rejoin}} \leftarrow 0.05$
    else
      $\lambda_{\mathit{leave}} \leftarrow 0, \quad \lambda_{\mathit{rejoin}} \leftarrow 0$ ▷ Stable peers never leave
    end if
    if $\omega_p =$ online **and** $\mathit{Uniform}(0,1) < \lambda_{\mathit{leave}}$ **then**
      $\omega_p \leftarrow$ offline
      Clear $p.\mathcal{M}$, $p.\mathcal{L}$ ▷ Mesh connections lost
    else if $\omega_p =$ offline **and** $\mathit{Uniform}(0,1) < \lambda_{\mathit{rejoin}}$ **then**
      $\omega_p \leftarrow$ online
      Clear $p.\mathit{seen}$, $p.\mathit{mcache}$ ▷ State reset on rejoin
    end if
  end for
end procedure

Algorithm 11 Bandwidth Enforcement

procedure ENFORCEBANDWIDTH(t)
  for all message m from sender s scheduled for delivery at tick $t+1$ **do**
    $\mathit{size} \leftarrow |m|$ ▷ Payload or control message size
    if $s.\mathit{tokens} < \mathit{size}$ **then**
      Remove m from schedule ▷ Drop: insufficient bandwidth
    else
      $s.\mathit{tokens} \leftarrow s.\mathit{tokens} - \mathit{size}$
      if m is payload **then**
        $s.B^{\texttt{Payload}} \leftarrow s.B^{\texttt{Payload}} + \mathit{size}$
      else
        $s.B^{\texttt{Ctrl}} \leftarrow s.B^{\texttt{Ctrl}} + \mathit{size}$
      end if
    end if
  end for
end procedure

Algorithm 12 Gossip Emission (`IHAVE`)

procedure EMITGOSSIP(t)
  for all online peer p **do**
    $ids \leftarrow$ message IDs in $p.mcache$ from last 3 heartbeats
    if $ids = \emptyset$ **then**
      continue
    end if
    $k \leftarrow \min(D_{lazy}, |p.\mathcal{L}|)$
    $targets \leftarrow$ sample k peers uniformly from $p.\mathcal{L}$
    for all $q \in targets$ **do**
      if $\omega_q = \mathsf{online}$ **then**
        Schedule `IHAVE`(ids) from p to q at tick $t + \text{DELAY}(p, q)$
      end if
    end for
    Advance $p.mcache$ window ▷ Expire old entries
  end for
end procedure

Algorithm 13 Process `IHAVE`/ `IWANT`

procedure PROCESSIHAVE(p, ids, $sender$)
  $missing \leftarrow ids \setminus p.seen$
  if $missing \neq \emptyset$ **then**
    Schedule `IWANT`($missing$) from p to $sender$ at tick $t + \text{DELAY}(p, sender)$
  end if
end procedure
procedure PROCESSIWANT(p, ids, $requester$)
  for all $mid \in ids$ **do**
    if $mid \in p.mcache$ **then**
      $m \leftarrow p.mcache[mid]$
      Schedule delivery of m from p to $requester$ at tick $t + \text{DELAY}(p, requester)$
    end if
  end for
end procedure

Algorithm 14 Mesh Maintenance (GRAFT/ PRUNE)

procedure MAINTAINMESH(t)
  for all online peer p **do**
    $p.\mathcal{M} \leftarrow \{q \in p.\mathcal{M} : \omega_q = \text{online}\}$ ▷ Remove offline peers
    if $|p.\mathcal{M}| < D - 2$ **then** ▷ Below D_{lo}: graft
      $k \leftarrow D - |p.\mathcal{M}|$
      $candidates \leftarrow \{q \in p.table \setminus p.\mathcal{M} : \omega_q = \text{online}\}$
      $new \leftarrow$ sample $\min(k, |candidates|)$ peers from $candidates$
      for all $q \in new$ **do**
        $p.\mathcal{M} \leftarrow p.\mathcal{M} \cup \{q\}$
        Schedule GRAFT(p) to q at tick $t + \text{DELAY}(p, q)$
      end for
    else if $|p.\mathcal{M}| > D + 2$ **then** ▷ Above D_{hi}: prune
      $k \leftarrow |p.\mathcal{M}| - D$
      $excess \leftarrow$ sample k peers from $p.\mathcal{M}$
      for all $q \in excess$ **do**
        $p.\mathcal{M} \leftarrow p.\mathcal{M} \setminus \{q\}$
        Schedule PRUNE(p) to q at tick $t + \text{DELAY}(p, q)$
      end for
    end if
    ▷ Update lazy mesh from remaining non-eager peers
    $p.\mathcal{L} \leftarrow$ sample D_{lazy} peers from $\{q \in p.table \setminus p.\mathcal{M} : \omega_q = \text{online}\}$
  end for
end procedure

Algorithm 15 Process GRAFT/ PRUNE

procedure PROCESSGRAFT(p, $requester$)
  if $requester \in p.table$ **and** $\omega_{requester} = \text{online}$ **then**
    $p.\mathcal{M} \leftarrow p.\mathcal{M} \cup \{requester\}$
    $p.\mathcal{L} \leftarrow p.\mathcal{L} \setminus \{requester\}$
  end if
end procedure
procedure PROCESSPRUNE(p, $requester$)
  $p.\mathcal{M} \leftarrow p.\mathcal{M} \setminus \{requester\}$
end procedure

Algorithm 16 Compute Final Metrics

function ComputeMetrics
  ▷ Delivery set for message m: peers that received it
  $R_m \leftarrow \{p \in V : m \in p.delivered\}$
  ▷ Delivery rate: fraction of all peers that received each message
  $\bar{\delta} \leftarrow \frac{1}{|M|} \sum_{m \in M} \frac{|R_m|}{|V|}$
  ▷ Total bandwidth
  $B_{total} \leftarrow \sum_{p \in V} \left(p.B^{\texttt{Payload}} + p.B^{\texttt{Ctrl}} \right)$
  ▷ Bytes per delivery
  $deliveries \leftarrow \sum_{m \in M} |R_m|$
  $\beta \leftarrow B_{total} / deliveries$
  ▷ Normalized cost
  $\beta_{norm} \leftarrow \beta / P$ ▷ $P = 1536$ bytes
  ▷ p99 latency over all successful deliveries
  $L \leftarrow \{(t_{m,p} - t_m) \cdot \Delta t : m \in M,\, p \in R_m\}$
  $p99 \leftarrow Percentile_{99}(L)$
  return $(\bar{\delta}, \beta_{norm}, p99)$
end function

8.2.6 Pareto Dominance Analysis

Definition 8.2.38 (Pareto Dominance)

Configuration A *dominates* configuration B on delivery rate and bandwidth cost if $\delta_A \geq \delta_B$ and $(\beta/P)_A \leq (\beta/P)_B$, with at least one strict inequality.

We exclude latency from the dominance computation because Floodsub achieves the lowest latency (0.4 s p99). Including latency would prevent any configuration from strictly dominating Floodsub, obscuring the clear delivery-cost advantage of Gossipsub. Latency is instead encoded as marker shape in Figure 8.3.3 for visual inspection.

For the dominance plot, we fix the churn level at $c = 0.2$, the empirically grounded value from Kiffer *et al*, and extract metrics directly for each (D, D_{lazy}) configuration using Algorithm 17.

Algorithm 17 Filter Results at Fixed Churn Level ($c = 0.2$)

Require: Raw results $\mathcal{R} = \{(D, D_{lazy}, c, \bar{\delta}, \beta_{norm}, p99)\}$ for all configurations
Ensure: Filtered results $\mathcal{A}$ at $c = 0.2$
$\mathcal{A} \leftarrow \emptyset$
for all $r \in \mathcal{R}$ **do**
  if $r.c = 0.2$ **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{(r.D,\ r.D_{lazy},\ r.\bar{\delta},\ r.\beta_{norm},\ r.p99)\}$
  end if
end for
return $\mathcal{A}$

Algorithm 18 computes the 2D Pareto frontier over delivery rate and cost. Latency is retained in $\mathcal{A}$ for visualization but is not used in the dominance check.

Algorithm 18 Compute 2D Pareto Frontier

Require: Filtered results $\mathcal{A} = \{(D, D_{lazy}, \bar{\delta}, \beta_{norm}, p99)\}$ at $c = 0.2$
Ensure: Pareto-optimal set $\mathcal{P} \subseteq \mathcal{A}$

function DOMINATES(A, B) ▷ Does A dominate B on delivery rate and cost?
  return $(\bar{\delta}_A \geq \bar{\delta}_B) \wedge (\beta_{norm,A} \leq \beta_{norm,B}) \wedge \big((\bar{\delta}_A > \bar{\delta}_B) \vee (\beta_{norm,A} < \beta_{norm,B})\big)$
end function
$\mathcal{P} \leftarrow \mathcal{A}$
for all $A \in \mathcal{A}$ **do**
  for all $B \in \mathcal{A} \setminus \{A\}$ **do**
    if DOMINATES(B, A) **then**
      $\mathcal{P} \leftarrow \mathcal{P} \setminus \{A\}$
      break
    end if
  end for
end for
return $\mathcal{P}$

8.2.7 Simulation Parameters

Table 8.3 lists all simulation parameters used in our experiments.

8.2.8 Validation Against Published Measurements

To validate our simulator, we compare against Protocol Labs' Gossipsub v1.1 evaluation [74], which tested a 1000-node network with $D = 8$, $D_{lazy} = 8$, and no churn. We use a fixed 25 ms per-hop delay to match Protocol Labs' 50 ms RTT assumption. Our validation aims to establish the correctness of Gossipsub's dissemination dynamics and the relative ordering of configurations, rather than to predict exact absolute performance in production deployments. We therefore validate against published testbed results under controlled conditions and emphasize qualitative agreement and dominance relationships, which are robust to simulation artifacts.

Theoretical Latency Analysis

We first derive expected latency bounds from graph-theoretic properties of the mesh overlay.

Average shortest-path length. The eager mesh forms a random regular graph with degree D. For such graphs, the average shortest-path length (ASPL) scales logarithmically:

$$\text{ASPL} \approx \frac{\ln n}{\ln D} = \frac{\ln 1000}{\ln 8} \approx 3.32 \text{ hops} \tag{8.10}$$

Table 8.3: Simulation parameter values

Parameter	**Symbol**	**Value**
Network		
Number of peers	n	1000
Peer table size		$[40, 80]$
Validators per peer	v	1
Timing		
Tick duration	Δt	0.1 s
Heartbeat interval		1.0 s
Warmup period	T_w	2000 ticks
Measurement period	T	10000 ticks
Protocol		
Mesh degree	D	$\{2, 4, \ldots, 14\}$
Gossip degree	D_{lazy}	$\{1, 3, \ldots, 33\}$
Gossip history		3 heartbeats
Messages		
Payload size	P	1536 bytes
Message ID size		32 bytes
IHAVE overhead		32 bytes
IWANT overhead		16 bytes
GRAFT/PRUNE size		100 bytes
Bandwidth		
Global upload cap	B	25 Mbps
Token bucket size		$B/8$ bytes
Refill rate		$B \cdot \Delta t/8$
Churn		
Churny fraction	c	$\{0, 0.1, \ldots, 0.3\}$
Churny leave rate	λ^c_{leave}	0.01 / tick
Churny rejoin rate	λ^c_{rejoin}	0.05 / tick
Stable leave rate	λ^s_{leave}	0
Replication		
Seeds per config		3

Table 8.4: Validation against Protocol Labs Gossipsub evaluation (1000 nodes, $D = 8$, $D_{lazy} = 8$, 0% churn, 10 ms ticks)

Metric	Ours	Protocol Labs	Notes
Delivery rate	99.7%	100%	matches
p50 latency	0.240 s	0.100 s	2.4× (tick overhead)
$p99$ latency	0.390 s	0.165 s	2.4× (tick overhead)
Maximum latency	0.540 s	0.350 s	1.5×
ETH2 req. (<3 s)	✓	✓	Both satisfy

Empirical measurements from our simulation refine this to ASPL ≈ 3.0 hops, consistent with ProbeLab's DEthna measurements on the Goerli testnet (ASPL ≈ 2.7 for $\sim$1–2k nodes) [89].

Expected end-to-end latency. Given one way latency per-hop with mean μ_h and standard deviation σ_h, the expected end-to-end latency is:

$$\mu_{\text{e2e}} \approx \text{ASPL} \times \mu_h \tag{8.11}$$

The variance combines within-path variability and between-path hop-count variability:

$$\sigma^2_{\text{e2e}} \approx \text{ASPL} \times \sigma^2_h + \text{Var(hops)} \times \mu^2_h \tag{8.12}$$

Assuming Gaussian approximation, the 99th percentile is:

$$\text{p99} \approx \mu_{\text{e2e}} + 2.326 \times \sigma_{\text{e2e}} \tag{8.13}$$

For Protocol Labs' parameters ($\mu_h \approx 25$ ms from 50 ms RTT, negligible variance), predicted $p99$ is $3.0 \times 25\text{ms} \approx 75$ ms. Protocol Labs reports 165 ms, with the $\approx$2× difference attributable to protocol processing overhead (deserialization, validation, forwarding decisions).

Validation Results

Table 8.4 compares our simulator against Protocol Labs measurements. We test with per-hop delay of 25 ms (matching Protocol Labs' 50 ms RTT assumption) using 10 ms tick resolution.

Delivery rate. Our simulator achieves 99.7% delivery rate compared to Protocol Labs' 100%. This minor difference is an artifact of our measurement methodology: as discussed in Section 8.2.5, we compute delivery rate at the end of the measurement period T rather than waiting for full stabilization. Messages published near the end of the simulation may still be in transit when statistics are computed, yielding $\delta_{m,T} < \delta_m$. With sufficient settling time, we expect 100% delivery rate under zero churn, confirming that our implementation correctly captures Gossipsub's dissemination mechanics.

Latency. Absolute latencies are approximately 2.4× higher than Protocol Labs due to tick-based simulation overhead. With ASPL $\approx$ 3–4 hops and effective per-hop latency of $\sim$60 ms (25 ms network + tick overhead), predicted p50 is $3.5 \times 60\text{ms} = 210$ ms, consistent with our measured 240 ms. Protocol Labs' event-driven simulator avoids this overhead, yielding latencies closer to theoretical network delay. Crucially, both simulators satisfy

Ethereum's informal requirement of full attestation propagation within 3 seconds with substantial margin. The latency difference reflects simulation methodology rather than protocol behavior. Our discrete-time model (Algorithm 5) uses 100 ms ticks for the parameter sweep, introducing quantization overhead of $\lceil delay/\Delta t \rceil + 1$ ticks per hop; we reduce it by using 10 ms ticks for validation.

Implications for parameter sweep. Since delivery rate, our primary metric, matches Protocol Labs, and latency differences arise from consistent simulation overhead, relative comparisons across configurations remain valid. The parameter sweep (Section 8.2.8) uses 100 ms ticks for computational efficiency; all 320 configurations are evaluated under identical conditions, ensuring fair comparison.

Qualitative Trend Validation

Beyond absolute metric values, we validate that our simulation reproduces the qualitative behavior documented in prior work:

Monotonic delivery improvement with D: Protocol Labs observes that higher mesh degree improves delivery delivery, consistent with our Figure 8.3.1 (a) appearing later in the Results section.

Delivery rate–Bandwidth cost tradeoff: The evaluation report notes that increased connectivity comes at bandwidth cost, matching our finding in Figure 8.3.1 (b).

Gossip as recovery mechanism: Protocol Labs describes lazy pull (gossip) as enabling message recovery when mesh paths fail, consistent with our observation that D_{lazy} improves delivery under churn (Figure 8.3.2(a)).

Latency sensitivity to gossip: The report notes that gossip propagation adds latency (up to one heartbeat interval per hop), explaining the tail latency behavior we observe.

Limitations of Validation

We acknowledge several limitations in our validation approach:

Churn validation: No published measurements characterize Gossipsub behavior under controlled churn conditions comparable to our simulation. Our churn model is based on connection duration measurements from Kiffer *et al* [34], but delivery rate degradation under churn has not been validated against production data.

Scale gap: Our 1000-node simulation is substantially smaller than production Ethereum networks. While we verified that qualitative tradeoffs and the relative ordering of parameter configurations persist at larger scales, absolute metric values may differ.

Simplified protocol model: We omit peer scoring, multi-topic overlays, and Floodsub compatibility. These simplifications may affect behavior under adversarial conditions but should minimally impact benign churn scenarios.

Despite these limitations, the consistency between our simulation and Protocol Labs' testbed results, particularly for baseline latency, and the qualitative structure of delivery-bandwidth tradeoffs, provides confidence that our methodology captures the essential dissemination dynamics of Gossipsub.

8.3 Simulation Results

We now evaluate how Gossipsub parameters influence the delivery—efficiency—latency trade-off defined in Section 8.2.5. Our analysis focuses on two protocol parameters: the mesh degree D, which controls eager-push connectivity, and the gossip degree D_{lazy}, which controls lazy dissemination. We simulate Floodsub (D=60, D_{lazy}=0) as a baseline representing pure eager-push flooding without gossip-based recovery. Since Floodsub is modeled via Gossipsub parameters, it inherits mesh maintenance behavior; rejoining peers establish mesh connections within one heartbeat interval (∼1 s). This approximates realistic Floodsub behavior, as establishing 60 TCP connections after rejoin would incur comparable latency. For each parameter configuration, we report results across all five churn regimes (0–30%) with error bars indicating the range across the other parameter dimension. This visualization reveals both typical behavior and the envelope of outcomes under varying network conditions. The separation is critical in churned networks, where churn-induced message loss can significantly impact protocol performance.

8.3.1 Effect of Mesh Degree

Figure 8.3.1 shows the impact of mesh degree D on delivery rate, bandwidth cost, and tail latency. Each line corresponds to a churn regime, with vertical bars indicating the range across gossip degrees D_{lazy}.

Delivery. Delivery rate improves with increasing D but exhibits strong diminishing returns, plateauing beyond $D \approx 8$. At 0% churn, all Gossipsub configurations achieve ∼100% delivery. At 10–20% churn, delivery remains above 99% for $D \geq 4$. At 30% churn, configurations with $D \geq 4$ maintain ∼99% delivery (98.8–99.2% depending on D_{lazy}), while Floodsub (D=60) degrades to 93.5%. The narrow error bars indicate that D_{lazy} has modest impact on delivery compared to D and churn level.

Efficiency. Normalized cost (β/P) grows monotonically with D. Floodsub incurs ∼60× overhead, while Gossipsub ranges from 3× at low D and churn to ∼200× at high D, D_{lazy}, and churn. The wide vertical bars reveal that D_{lazy} substantially impacts cost—higher gossip degrees increase bandwidth consumption significantly. Low-D configurations with minimal gossip achieve comparable delivery at substantially lower cost.

Latency. Tail latency decreases with increasing D, reflecting faster propagation in well-connected meshes. Floodsub achieves the lowest latency (∼0.3 s) due to pure eager-push with no gossip delays. Gossipsub latency ranges from ∼0.5–2 s at 0% churn to 5–8 s under high churn, with higher D reducing latency at all churn levels.

Key insight: Under non-zero churn (10–30%), Gossipsub dominates Floodsub on both delivery and cost; at 0% churn, Floodsub achieves 100% delivery that sparse Gossipsub meshes cannot match. Floodsub's consistent advantage across all churn levels is sub-second latency. Gossip-based recovery ($D_{lazy} \geq 1$) provides delivery that additional eager-push connections alone cannot achieve under churn.

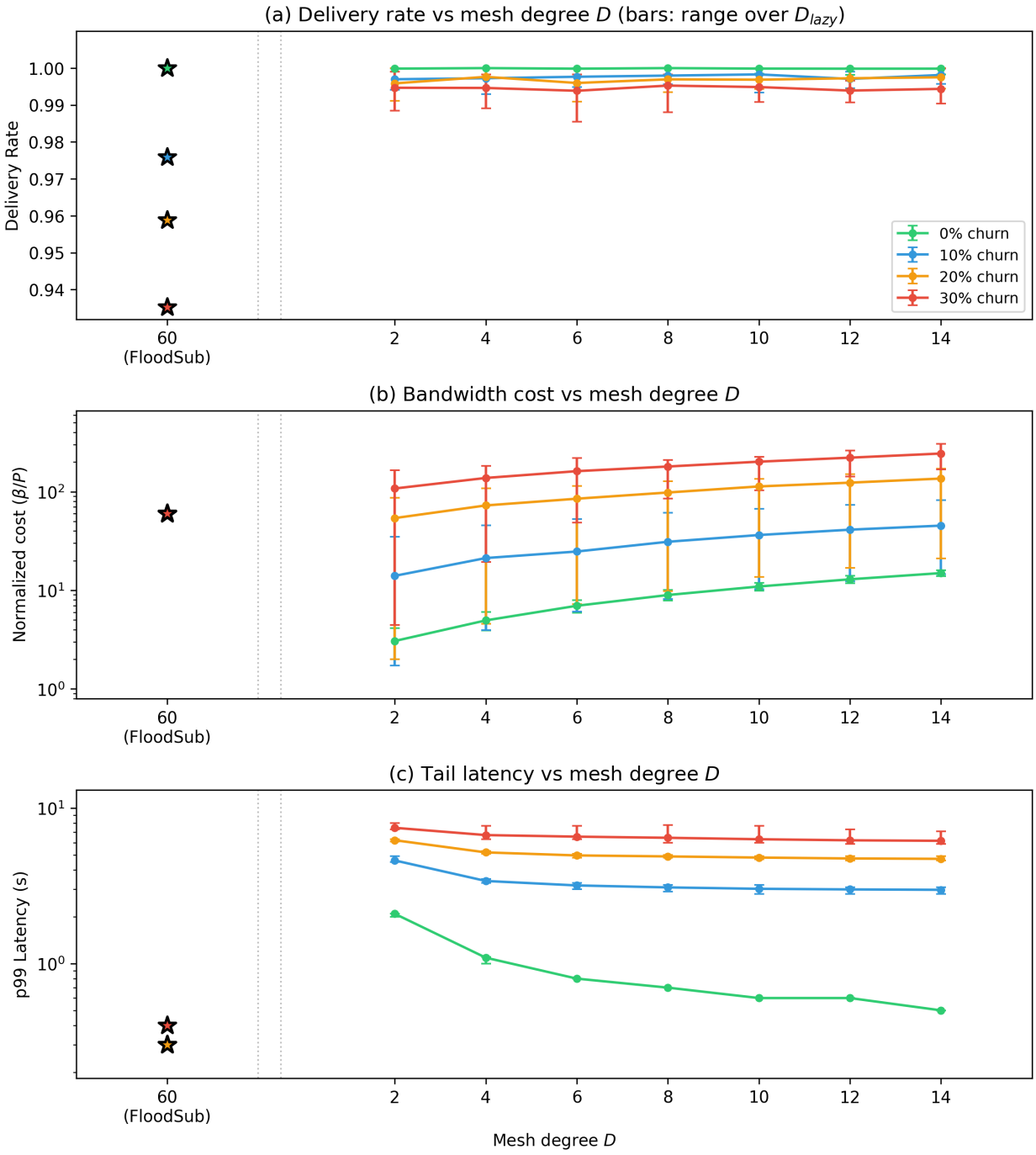


Figure 8.3.1: Effect of mesh degree D under varying churns. Each line represents a churn level (0–30%), plotting the mean across all D_{lazy} values; error bars show the range over $D_{lazy} \in \{1, 3, \ldots, 21\}$. Floodsub ($D$=60, D_{lazy}=0) shown as stars. (a) Delivery rate vs. D. (b) Normalized cost (β/P) vs. D. (c) Tail latency vs. D.

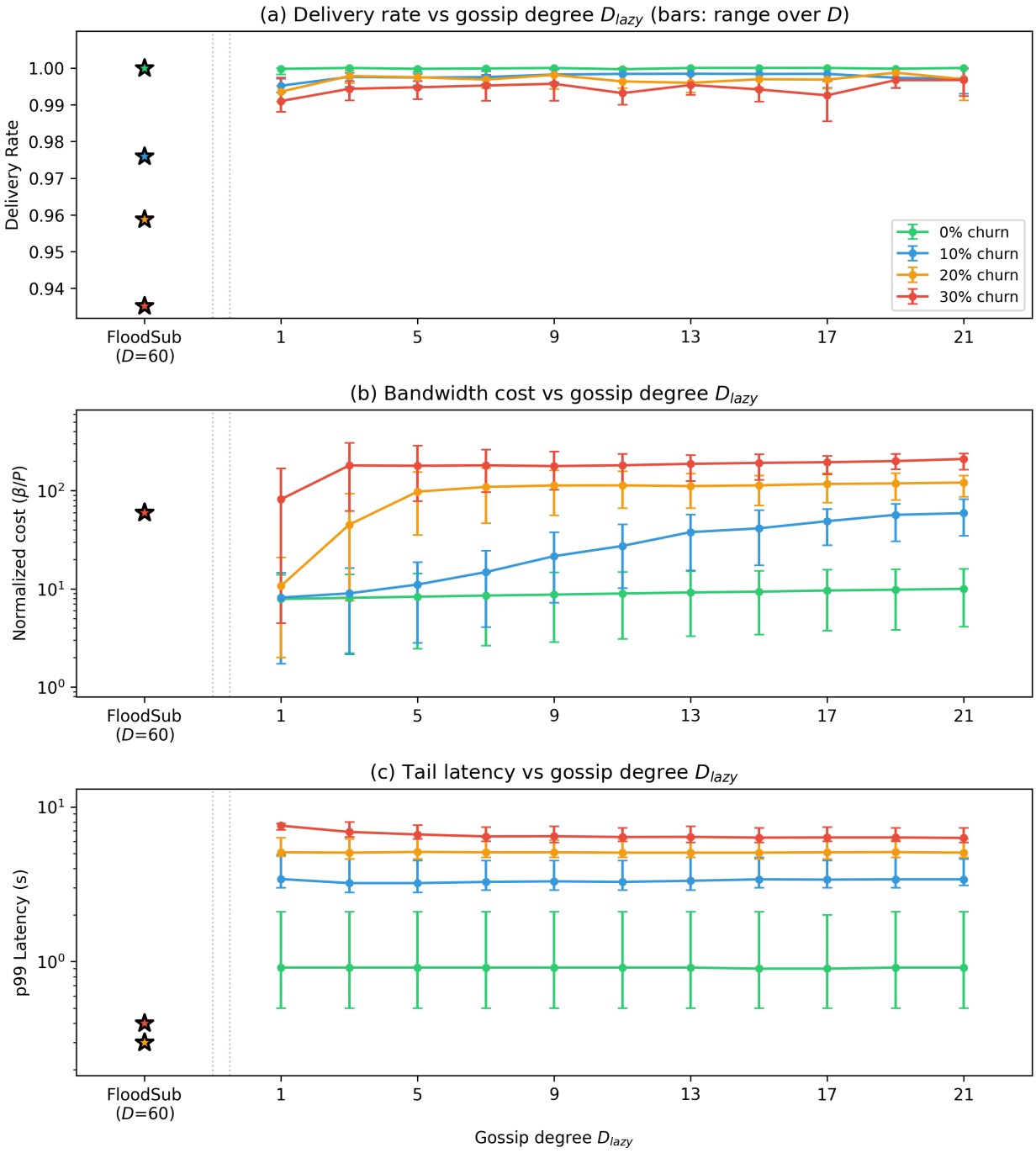


Figure 8.3.2: Effect of gossip degree D_{lazy} under varying churns. Each line represents a churn level (0–30%), plotting the mean across all D values; error bars show the range over $D \in \{2, 4, \ldots, 14\}$. Floodsub ($D$=60, D_{lazy}=0) shown as stars. (a) Delivery rate vs. D_{lazy}. (b) Normalized cost (β/P) vs. D_{lazy}. (c) Tail latency vs. D_{lazy}.

Table 8.5: Gossipsub configurations dominating Floodsub at each churn level. Minimal gossip (D_{lazy}=1) with sparse mesh exceeds Floodsub delivery at every churn level while achieving substantial cost savings.

Churn	Floodsub	Gossipsub	GS Delivery	Cost Savings
0%	100.0%	(2, 1)	100.0%	30×
10%	97.6%	(2, 1)	99.4%	35×
20%	95.9%	(2, 1)	99.2%	30×
30%	93.5%	(2, 1)	98.8%	13×

8.3.2 Effect of Gossip Degree

Figure 8.3.2 evaluates the impact of gossip degree D_{lazy}. Each line corresponds to a churn regime, with vertical bars indicating the range across mesh degrees $D \in \{2, 4, \ldots, 14\}$.

Delivery. Minimal gossip (D_{lazy}=1) achieves ∼100% delivery at 0% churn and >99% delivery at 20% churn, exceeding Floodsub's 95.9% at the same churn level. Increasing D_{lazy} provides negligible improvement: at 20% churn, delivery rises by only 0.3 percentage points from D_{lazy}=1 to D_{lazy}=21 (99.4% to 99.7%). The wide vertical bars indicate that D has substantial impact on delivery, particularly under high churn, while D_{lazy} has almost no effect beyond the minimal value.

Efficiency. Normalized cost increases sharply with D_{lazy}, from 8–82× at D_{lazy}=1 to 10–210× at D_{lazy}=21 depending on churn. This reflects IHAVE/IWANT control message overhead. High-D_{lazy} configurations under churn exceed Floodsub's ∼60× cost while providing negligible delivery improvement.

Latency. Tail latency shows high variability under churn, with D_{lazy} having modest direct effect. At 0% churn, latency remains below 2 s. Under 10–30% churn, latency ranges from ∼3 s to ∼8 s depending primarily on D (shown by wide vertical bars). Floodsub achieves the lowest latency (∼0.3 s) due to pure eager-push.

8.3.3 Best-effort Trade-offs and Dominance

Figure 8.3.3 plots delivery against normalized cost at 20% churn (the empirically grounded churn level from Kiffer *et al* [34]), with marker shape encoding $p99$ latency. Floodsub achieves the lowest latency (0.4 s), so it is not Pareto-dominated in the full three-dimensional objective space. However, on the delivery—cost plane at this churn level, it lies below the frontier: multiple Gossipsub configurations achieve both higher delivery and lower cost simultaneously.

Gossipsub dominates Floodsub on delivery and cost under churn. Floodsub (D=60, D_{lazy}=0) achieves 95.9% delivery at 60× cost at 20% churn. Every D_{lazy}=1 configuration (blue markers) exceeds this delivery at lower cost. Table 8.5 shows representative configurations: at 10% churn, $(2, 1)$ achieves 99.4% delivery at 35× lower cost; at 20% churn, $(2, 1)$ achieves 99.2% at 30× lower cost.

Floodsub's only advantage is sub-second latency. D_{lazy}=1 configurations (blue markers) fall in the 1–5 s p99 latency tier. (D=8, D_{lazy}=3) matches Ethereum's 4.9 s latency ex-

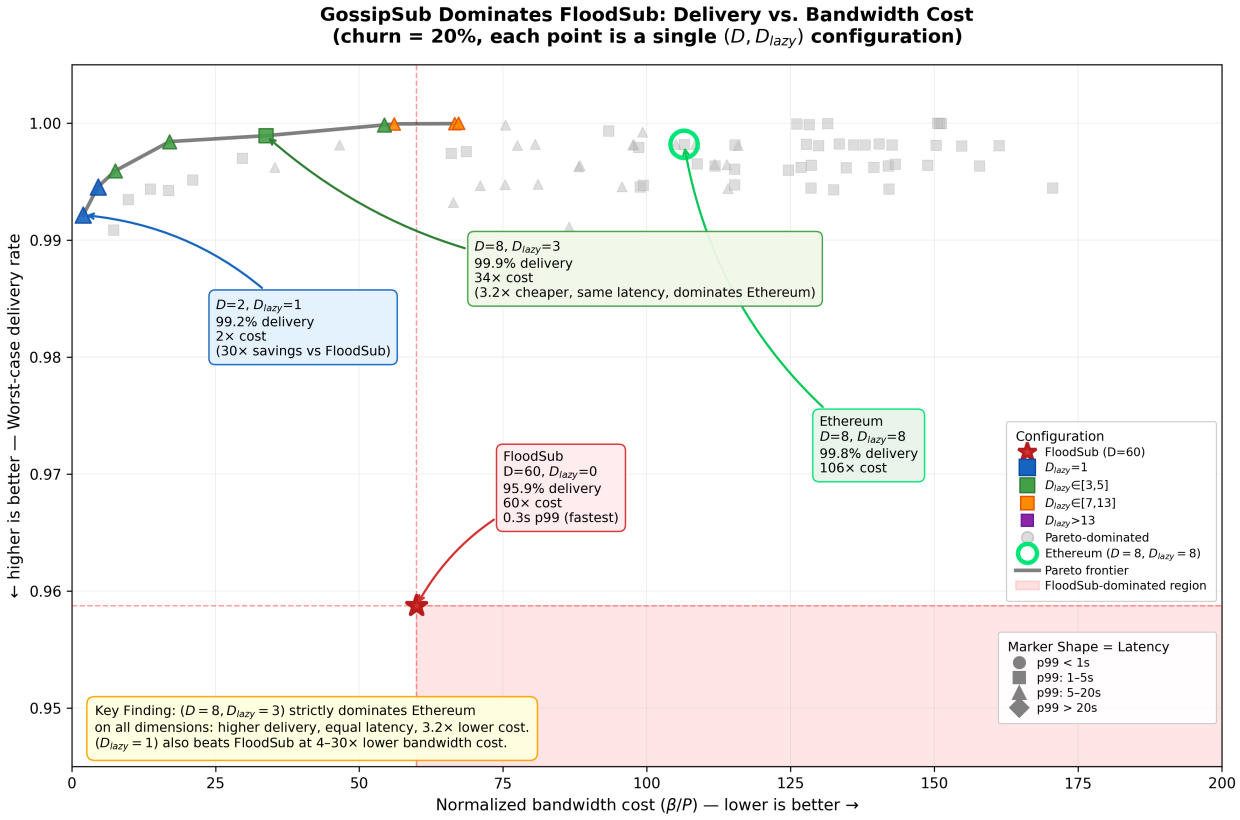


Figure 8.3.3: Pareto dominance: delivery vs. normalized cost at 20% churn (empirically grounded churn level from Kiffer *et al* [34]). Each point is a (D, D_{lazy}) configuration. Marker color indicates D_{lazy}; shape indicates $p99$ latency tier. Floodsub (D=60, red star) achieves 95.9% delivery at 60× cost. D_{lazy}=1 configurations (blue) achieve higher delivery at 2–21× cost. Red region: configurations inferior to Floodsub on both delivery and cost.

actly while beating it on both delivery and cost; (D=8, D_{lazy}=1) achieves 4.8 s— slightly faster than Ethereum. Both are well within Ethereum's 12 s slot time. Floodsub's 0.3 s latency matters only for applications requiring sub-second propagation, at 60× the bandwidth cost.

8.3.4 Tradeoff Against Ethereum's Deployed Configuration

Ethereum deploys Gossipsub with mesh degree D=8 and gossip degree D_{lazy}=8. We compare this configuration against the Pareto frontier at 20% churn—the empirically calibrated churn level from Kiffer *et al* [34].

At 20% churn, Ethereum's configuration (D=8, D_{lazy}=8) achieves 99.8% delivery and 4.9 s p99 latency at 106× normalized bandwidth cost. Figure 8.3.3 shows that Lean Gossip configurations with $D_{lazy} \leq 3$ strictly dominate Ethereum on all three objectives simultaneously—higher delivery, lower or equal latency, and lower cost.

(D=8, D_{lazy}=3) is the clearest example: it achieves 99.9% delivery and 4.9 s p99 latency at 34× cost—strictly better than Ethereum on delivery and cost, at identical latency, for a 3.2× bandwidth saving. (D=14, D_{lazy}=3) further improves latency to 4.6 s at 99.9% delivery, though with a more modest 1.1× cost saving. The leaner (D=8, D_{lazy}=1) pushes cost down to 9.8×—nearly 11× cheaper than Ethereum—while achieving slightly better latency (4.8 s) at the cost of 0.5 percentage points of delivery.

This establishes that Ethereum's deployed gossip degree is over-provisioned for benign churn scenarios: delivery scales primarily with mesh degree D, while increasing D_{lazy} beyond 1 incurs steep bandwidth cost for only 0.3 percentage points of additional delivery across the full D_{lazy} range.

Thus, Lean Gossip occupies a more favorable position on the delivery—bandwidth Pareto frontier than Ethereum's deployed configuration at empirically grounded churn levels.

8.4 Discussion

Our results reveal three structural insights about gossip-based dissemination under churn.

8.4.1 Why Floodsub Lies Off the Pareto Frontier Under Churn

Floodsub attempts delivery through network-layer redundancy: maintaining 60 eager-push connections to maximize delivery probability. However, under churn, if a message fails to reach a node because its mesh peers are offline, there is no recovery mechanism. Gossipsub employs protocol-layer repair via IHAVE/IWANT exchanges, enabling missed messages to be recovered even with sparse meshes. As a result, under non-zero churn Floodsub delivers fewer messages at dramatically higher bandwidth cost. At 0% churn, Floodsub achieves 100% delivery that sparse Gossipsub meshes cannot match. Its only consistent advantage across all churn levels is sub-second latency.

8.4.2 Why Lean Gossip Strictly Dominates Ethereum's Configuration

Ethereum deploys D=8, D_{lazy}=8, reflecting a conservative balance between redundancy and repair. Our Pareto analysis reveals that this balance is unnecessary: $D_{lazy} \leq 3$ configurations strictly dominate Ethereum on all three performance objectives at 20% churn. (D=8, D_{lazy}=3) achieves higher delivery (99.9% vs. 99.8%), equal latency (4.9 s), and 3.2× lower cost. The leaner (D=8, D_{lazy}=1) reduces cost to nearly 11× below Ethereum with slightly better latency, at the cost of only 0.5 percentage points of delivery. Delivery scales primarily with mesh connectivity (D), not gossip fanout (D_{lazy}); reducing D_{lazy} from 8 to 3 loses nothing and saves 3× in bandwidth.

8.4.3 Recovery Dominates Redundancy

A central conceptual contribution of this work is identifying that gossip-based recovery—not eager-push redundancy—is the dominant mechanism for delivery under churn.

Increasing D beyond approximately 8 yields diminishing delivery returns. Increasing D_{lazy} beyond 1 sharply increases bandwidth cost for negligible delivery benefit: at 20% churn, delivery improves by only 0.3 percentage points across the full range $D_{lazy} \in [1, 21]$, while cost increases by more than 10×. The true tradeoff is therefore latency versus cost—not delivery versus cost. Sparse meshes combined with minimal gossip recovery dominate brute-force flooding in the delivery—bandwidth plane under realistic churn.

8.4.4 Practical Recommendations

Based on our simulations, we offer guidance for Gossipsub deployments.

For delivery-critical applications. Use $D \geq 8$ with D_{lazy}=1. These achieve approximately 98.8% delivery even at 30% churn at lower cost than Floodsub (Table 8.6).

For bandwidth-constrained environments. Configurations with $D_{lazy} \leq 3$ strictly dominate Ethereum on all performance dimensions and achieve delivery exceeding Floodsub at 3–30× lower cost. We recommend D=8, D_{lazy}=3 as the primary configuration: it strictly beats Ethereum on all three metrics at 3.2× lower cost. For the most bandwidth-constrained deployments, D=8, D_{lazy}=1 reduces cost to nearly 11× below Ethereum with only 0.5 percentage points of delivery reduction. Note that at 0% churn, Floodsub's 100% delivery cannot be matched by sparse configurations.

For latency-critical applications. If sub-second propagation is required, Floodsub (D=60, D_{lazy}=0) achieves 0.3 s $p99$ latency—but at 60× cost and 4% lower delivery at 20% churn.

Ethereum's parameters. Ethereum's deployed configuration (D=8, D_{lazy}=8) achieves 100% delivery at 0% churn and ∼99.8% at 20% churn; our results show that (D=8, D_{lazy}=3) strictly dominates these parameters—matching or exceeding all three metrics at 3.2× lower cost (Table 8.7).

Table 8.6: Representative configurations across the reliability-efficiency-latency spectrum (worst-case across churn regimes). Gossipsub configurations with D_{lazy}=1 achieve higher delivery than Floodsub at lower cost; Floodsub's only advantage is latency.

D	D_{lazy}	Delivery	p99 Latency	Cost (β/P)
2	1	98.8%	7.6 s	4×
4	1	98.9%	7.7 s	19×
8	1	98.8%	7.8 s	85×
12	1	99.2%	7.3 s	143×
14	1	99.0%	7.1 s	168×
Floodsub reference (D=60, D_{lazy}=0)				
60	0	93.5%	0.4 s	60×

Table 8.7: Delivery rate by churn level for Gossipsub (D=8, D_{lazy}=1) compared to Floodsub (D=60, D_{lazy}=0). Gossipsub achieves higher delivery at every churn level >0%.

Churn	**GS(8,1) Delivery**	**GS(8,1) p99**	**Floodsub Delivery**	**Floodsub p99**
0%	100.0%	0.7 s	100.0%	0.3 s
10%	99.6%	3.2 s	97.6%	0.3 s
20%	99.3%	4.8 s	95.9%	0.3 s
30%	98.8%	7.8 s	93.5%	0.4 s

Gossip explosion under churn. At D=8 with D_{lazy}=21 and 30% churn, total traffic reaches 170.5 GB—3× higher than D_{lazy}=1 (67.9 GB)—while improving delivery by only 0.8 percentage points (98.8% to 99.7%) relative to D_{lazy}=1 (Table 8.8). This occurs because churning peers repeatedly request messages via IHAVE/IWANT that never complete. Protocol designers should consider gossip backoff under detected churn.

8.5 Limitations and Future Work

Our analysis has several limitations. First, our churn model assumes uniform leave/join probabilities, whereas real networks may exhibit correlated churn (*e.g.*, geographic outages affecting multiple peers simultaneously). No published measurement study currently characterizes correlated churn in Gossipsub deployments; our modular simulator is designed to accommodate alternative churn models as such data becomes available. Second, we do not model adversarial behavior; our parameter recommendations optimize for benign churn, and attack resilience under these parameters remains an open question. Protocol Labs evaluated Gossipsub v1.1's resilience against Sybil and Eclipse attacks [74, 75], and a Least Authority security audit identified further concerns with the peer scoring mechanism [50]. More recently, Kumar *et al* [39, 41] used formal model-driven analysis to identify correctness problems with Gossipsub's adversarial safeguards in Ethereum configurations, showing that certain parameter settings can be exploited to cause Eclipse attacks or network partitions. Whether our recommended Lean Gossip parameters are equally or more vulnerable to such

Table 8.8: Diminishing returns of gossip degree (D=8, 30% churn). Increasing D_{lazy} from 1 to 21 improves delivery by only 0.8% points while increasing traffic 3×.

D_{lazy}	Delivery	Δ Delivery	Total Traffic	Traffic Mult.
1	98.8%	—	67.9 GB	1×
5	99.3%	+0.5 pp	144.4 GB	2×
9	99.5%	+0.7 pp	145.2 GB	2×
13	99.6%	+0.8 pp	150.8 GB	2×
17	99.6%	+0.8 pp	160.0 GB	2×
21	99.7%	+0.8 pp	170.5 GB	3×

attacks is ongoing research.

Future work should extend this analysis to multi-topic scenarios, incorporate peer scoring mechanisms from Gossipsub v1.1 [49], and validate recommendations through testbed experiments or instrumentation of production networks. Additionally, our Pareto optimization framework could be applied to emerging protocol variants such as Gossipsub v2.0 [70], which introduces probabilistic lazy forwarding within the mesh itself. Finally, integrating our parameter selection methodology into adaptive protocols that dynamically adjust D and D_{lazy} based on observed network conditions represents a promising direction.

8.6 Bibliographic Notes

One of the earliest investigations into P2P communication was conducted by Paul Baran [4] during the late 1950s, who proposed flooding as a mechanism for building resilient, low-data-rate networks capable of withstanding both equipment failures and intentional disruptions. He quantified connectivity vs. survivability by simulating random node failures and measuring the percentage of the network that remained reachable. The first challenge that arises in naive flooding is redundancy: the same message may be received multiple times by the same node, leading to bandwidth waste and network congestion. This issue was formalized in the broadcast storm problem by Ni *et al* [66] who identified the three core causes: contention, collision, and redundant rebroadcasts, through simulations in Mobile Ad hoc NETworks (MANETs). Gopal and Kutten [22] proposed protocols like BASIC and RELIEF that achieve efficient broadcast in networks by relying on timing guarantees: for example, nodes forward a message for a bounded duration before halting. While efficient, such approaches raise concerns about reliability of message delivery. More recently, selective rebroadcast schemes [78] reconsider the conventional suppression approach by actively promoting controlled redundancy. Rather than minimizing rebroadcasts, these schemes encourage additional transmissions to enhance reliability in lossy or highly dynamic network environments.

Lynch's study on reliable broadcast in networks with non-programmable servers [21] presents an early and influential algorithm for disseminating messages, explicitly analyzing the trade-offs between reliability and message transmission costs. While links may fail or recover dynamically, the protocol assumes that the network remains continuously connected. Beyond this work, Lynch and Vaandrager developed a refinement framework for verifying

distributed systems using forward and backward simulations. Their untimed framework [52] provides techniques for relating concrete implementations to abstract specifications via trace inclusion, enabling modular correctness proofs for asynchronous message-passing protocols. This framework was later extended to timing-based systems [51, 53], introducing simulation relations capable of reasoning about real-time constraints, timed actions, and clocks.

Parallel to these classical reliable-broadcast results, a substantial body of work developed probabilistic epidemic or gossip-based dissemination. Demers *et al* [16] demonstrated that a fanout of $\log(n)$ is sufficient to spread messages to nearly all nodes in $\mathcal{O}(\log(\backslash))$ rounds with high probability, assuming uniform random peer selection. Analytical models [5, 20] of protocols where nodes periodically exchange information with randomly chosen peers showed that coverage increases exponentially with fanout. These protocols trade deterministic guarantees for simplicity, resilience, and scalable expected performance, and they form the conceptual foundation underlying modern P2P publish-subscribe (pubsub) protocols such as Gossipsub.

Previous work by Cristo *et al* [15] presents a systematic evaluation to characterize and explain Gossipsub's behavior across several axes when integrated with XRP's consensus layer. The study examines nine key dimensions—including message propagation delay, mesh stability, scoring dynamics, resilience under churn, fault tolerance, and adversarial conditions to understand performance and reliability trade-offs.

Orthogonally, Vyzovitis *et al* [74, 75, 83] presented a detailed analysis of the resilience of Gossipsub implementations against Sybil and Eclipse attacks. On the contrary, Kumar *et al* [41] used formal model driven analysis to show that there exist Ethereum 2.0 configurations of Gossipsub prone to sybil attacks that can lead to eclipse or even network partitions.

Part IV

Conclusion

Chapter 9

Conclusion

This dissertation developed a principled framework for analyzing gossip-based pubsub protocols through compositional refinement, property-driven counterexample generation, and simulation-guided optimization. By studying Floodsub, Meshsub, and Gossipsub in succession, we showed how increasingly sophisticated protocol mechanisms can be related back to a simple broadcast specification, and how deviations from that specification can be precisely identified and explained.

At the heart of this work is a uniform dissemination guarantee: once a message is committed at its source, every correct (non-faulty, active, and subscribed) peer eventually receives it. We first proved this guarantee for Broadcastsub, then lifted it compositionally to Floodsub, and finally to Meshsub via probabilistic skipping refinement. These results demonstrate that selective mesh-based forwarding preserves semantic dissemination guarantees while enabling bandwidth reduction, provided explicit fairness, gossip assumptions and temporal connectedness conditions hold.

Crucially, the refinement proofs expose the structural role of mesh degree D and gossip fanout D_{lazy} as parameters governing the extent of probabilistic stuttering and dissemination delay. Rather than treating performance and correctness as separate concerns, the analysis makes their interaction explicit: the same parameters that bound refinement also determine real-world trade-offs between latency and bandwidth efficiency.

Extending the analysis to Gossipsub v1.1 revealed a fundamental mismatch between control-plane scoring and dissemination invariants. Property-based testing and automated counterexample generation showed that peer scoring can violate behavioral consistency assumptions required for dissemination correctness. These counterexamples were not merely theoretical; they exposed concrete attack gadgets and informed public vulnerability disclosures. The framework therefore serves not only as a vehicle for certification, but also as a diagnostic tool for identifying semantic weaknesses in deployed systems.

Finally, simulation-based exploration of the Gossipsub parameter space demonstrated that the structural insights derived from refinement analysis translate directly into practical guidance. Pareto analysis over delivery, bandwidth, and latency confirmed that gossip-based recovery—not eager redundancy—is the dominant mechanism for robustness under churn. In this way, formal reasoning and empirical evaluation reinforce one another.

Taken together, this work advocates a layered methodology for distributed protocol analysis: establish strong semantic guarantees for core abstractions, lift those guarantees com-

positionally, test the compatibility of extensions, and use simulation to guide deployment decisions. Rather than attempting end-to-end verification of production-scale systems, this approach isolates correctness arguments where they are most effective while retaining the ability to uncover subtle dissemination failures in real-world protocols.

9.1 Future Work

Several directions remain open.

Refinement with scoring. A natural next step is to integrate peer scoring into the refinement framework. This would require modeling score evolution explicitly and characterizing conditions under which scoring preserves, rather than undermines, dissemination guarantees.

Adversarial churn and strategic behavior. Our analysis assumes benign churn and fair scheduling. Extending the models to incorporate adversarial churn patterns and strategic peer behavior would strengthen the connection between formal guarantees and security analysis.

Multi-topic and cross-layer interactions. We focused primarily on single-topic dissemination. Extending the formal and simulation models to capture cross-topic interactions, shared mesh dynamics, and resource contention remains an important step toward characterizing behavior at scale. More broadly, this work opens the door to hybrid analysis pipelines in which formal verification, counterexample generation, and empirical evaluation operate in concert. Such pipelines may provide a scalable path toward principled correctness reasoning for complex distributed systems.

To support transparency, verification, and future research, all artifacts developed as part of this dissertation are publicly available, as described below.

9.2 Artifact Availability

The mechanized models, refinement proofs, property-based test generators, counterexample artifacts, and simulation infrastructure developed in this dissertation are publicly available at: https://github.com/ankitku/gossipsub-refinement-verification.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 1991. doi:10.1016/0304-3975(91)90224-P.

[2] Ioannis Aekaterinidis and Peter Triantafillou. Peer-to-peer publish-subscribe systems. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi:10.1007/978-1-4614-8265-9_1221.

[3] Noran Azmy, Stephan Merz, and Christoph Weidenbach. A rigorous correctness proof for pastry. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, 2016. doi:10.1007/978-3-319-33600-8_5.

[4] Paul Baran. On distributed communications: I. introduction to distributed communications networks. Technical Report RM-3420-PR, RAND Corporation, 1964. URL https://www.rand.org/pubs/research_memoranda/RM3420.html. Accessed: 2025-06-08.

[5] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 1999. doi:10.1145/317785.317786.

[6] Paul Bonsma. Most balanced minimum cuts. *Discrete Applied Mathematics*, 2010. doi:https://doi.org/10.1016/j.dam.2009.09.010.

[7] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 1988. doi:10.1016/0304-3975(88)90098-9.

[8] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 2002.

[9] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. The "acl2" sedan theorem proving system. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011. doi:10.1007/978-3-642-19835-9_27.

[10] Harsh Raju Chamarthi. *Interactive Non-theorem Disproving*. PhD thesis, Northeastern University, 2016.

[11] Harsh Raju Chamarthi and Panagiotis Manolios. Automated specification analysis using an interactive theorem prover. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, pages 46–53. FMCAD Inc., 2011. URL http://dl.acm.org/citation.cfm?id=2157665.

[12] Harsh Raju Chamarthi, Dillinger Peter C., Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. 2011. doi:10.4204/EPTCS.70.1.

[13] Harsh Raju Chamarthi, Dillinger Peter C., and Panagiotis Manolios. Data Definitions in the ACL2 Sedan. *ACL2*, 2014.

[14] S. Coretti, Y. Gilad, and R. Pass. The generals' scuttlebutt: Byzantine-resilient gossip protocols. *Theoretical Computer Science*, 2022. doi:10.1145/3548606.3560638.

[15] Flaviene Scheidt de Cristo, Jorge Augusto Meira, Jean-Philippe Eisenbarth, and Radu State. A 9-dimensional analysis of gossipsub over the XRP ledger consensus protocol. In *NOMS 2024 IEEE Network Operations and Management Symposium*, 2024. doi:10.1109/NOMS59830.2024.10575688.

[16] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

[17] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 1974. doi:10.1145/361179.361202.

[18] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon, and J. Strother Moore. ACL2s: "the ACL2 sedan". In *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, 2007. doi:10.1016/j.entcs.2006.09.018.

[19] Ethereum Foundation. Ethereum consensus layer networking specification: Phase 0 p2p interface. Technical report, Ethereum Foundation, 2020. URL https://github.com/ethereum/consensus-specs/blob/master/specs/phase0/p2p-interface.md.

[20] A. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Computers*, 2003. doi:10.1109/TC.2003.1176982.

[21] Hector Garcia-Molina, Boris Kogan, and Nancy A. Lynch. Reliable broadcast in networks with nonprogrammable servers. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, 1988. URL https://groups.csail.mit.edu/tds/papers/Lynch/icdcs88.pdf.

[22] Ajei S. Gopal, Inder S. Gopal, and Shay Kutten. Fast broadcast in high-speed networks. *IEEE/ACM Transactions on Networking*, 1999. doi:10.1109/90.769773.

[23] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015. doi:10.1145/2815400.2815428.

[24] Lioba Heimbach, Yann Vonlanthen, Juan Villacis, Lucianna Kiffer, and Roger Wattenhofer. Deanonymizing ethereum validators: The p2p network has a privacy issue. In *USENIX Security Symposium*, 2025. URL https://arxiv.org/abs/2409.04366.

[25] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 2019. doi:10.1145/3338843.

[26] Mitesh Jain and Panagiotis Manolios. Skipping refinement. In *Computer Aided Verification*, 2015. doi:10.1007/978-3-319-21690-4_7.

[27] Mitesh Jain and Panagiotis Manolios. Proving skipping refinement with acl2s. In *International Workshop on the ACL2 Theorem Prover and Its Applications*, 2015. doi:10.4204/EPTCS.192.9.

[28] Mitesh Jain and Panagiotis Manolios. Local and compositional reasoning for optimized reactive systems. In *Computer Aided Verification*, 2019. doi:10.1007/978-3-030-25540-4_32.

[29] R. Kane, P. Manolios, and S.K. Srinivasan. Monolithic verification of deep pipelines with collapsed flushing. In *Proceedings of the Design Automation and Test in Europe Conference*, 2006. doi:10.1109/DATE.2006.244077.

[30] Matt Kaufmann and J Strother Moore. ACL2 homepage, 2022. URL https://www.cs.utexas.edu/users/moore/acl2/.

[31] Matt Kaufmann and J Strother Moore. Acl2 sources. https://github.com/acl2/acl2, 2023. Accessed 12 June 2023.

[32] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, July 2000. doi:10.1007/978-1-4757-3188-0.

[33] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000. doi:10.1007/978-1-4615-4449-4.

[34] Lucianna Kiffer, Asad Salman, Dave Levin, Alan Mislove, and Cristina Nita-Rotaru. Under the hood of the ethereum gossip protocol. In *Financial Cryptography and Data Security*. Springer, 2021.

[35] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC '10)*, 2010. doi:10.1145/1806689.1806760.

[36] Ankit Kumar. Artifact repository for "refinement based reasoning of p2p protocols is feasible and effective". https://github.com/ankitku/gossipsub-refinement-verification, 2026.

[37] Ankit Kumar and Panagiotis Manolios. Mathematical programming modulo strings. In *Formal Methods in Computer Aided Design, FMCAD*, 2021. doi:10.34727/2021/ISBN.978-3-85448-046-4_36.

[38] Ankit Kumar and Panagiotis Manolios. A formalization of the correctness of the floodsub protocol. *Electronic Proceedings in Theoretical Computer Science*, 2025. doi:10.4204/eptcs.423.9.

[39] Ankit Kumar, Max von Hippel, Panagiotis Manolios, and Cristina Nita-Rotaru. Verification of gossipsub in acl2s. In *International Workshop on the ACL2 Theorem Prover and Its Applications*, 2023. doi:10.4204/EPTCS.393.10.

[40] Ankit Kumar, Max von Hippel, Panagiotis Manolios, and Cristina Nita-Rotaru. Verification of gossipsub in acl2s. In *International Workshop on the ACL2 Theorem Prover and Its Applications*, 2023. doi:10.4204/EPTCS.393.10.

[41] Ankit Kumar, Max von Hippel, Panagiotis Manolios, and Cristina Nita-Rotaru. Formal model-driven analysis of resilience of gossipsub to attacks from misbehaving peers. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, 2024. doi:10.1109/SP54263.2024.00017.

[42] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 1978. doi:10.1016/0376-5075(78)90045-4.

[43] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

[44] João Leitão, José Pereira, and Luís E. T. Rodrigues. Epidemic broadcast trees. In *Symposium on Reliable Distributed Systems (SRDS)*, 2007. doi:10.1109/SRDS.2007.27.

[45] Kai Li, Yuzhe Tang, Jiaqi Chen, Yibo Wang, and Xianghong Liu. TopoShot. In *Internet Measurement Conference*, 2021. doi:10.1145/3487552.3487814.

[46] libp2p. gossipsub v1.0: An extensible baseline pubsub protocol. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md, 2020. GitHub Repository - libp2p specifications.

[47] libp2p. gossipsub v1.1: Extensions for attack resistance. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md, 2020. GitHub Repository - libp2p specifications.

[48] libp2p. gossipsub v1.0: An extensible baseline pubsub protocol. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md, 2020. GitHub Repository - libp2p specifications.

[49] libp2p. gossipsub v1.1: Extensions for attack resistance. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md, 2020. GitHub Repository - libp2p specifications.

[50] Dylan Lott. Audit of Gossipsub v1.1 protocol design + implementation for protocol labs. https://leastauthority.com/blog/audit-of-gossipsub-v1-1-protocol-design-implementation-for-protocol-labs/, 2020. Accessed 3 March 2022.

[51] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations for timing-based systems. In *Real-Time: Theory in Practice, REX Workshop, Mook, 1991, Proceedings*, 1991. doi:10.1007/BFB0032002.

[52] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 1995. doi:10.1006/INCO.1995.1134.

[53] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations, II: timing-based systems. *Inf. Comput.*, 1996. doi:10.1006/INCO.1996.0060.

[54] Panagiotis Manolios. Correctness of pipelined machines. In *Formal Methods in Computer-Aided Design, FMCAD*, 2000. doi:10.1007/3-540-40922-X_11.

[55] Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, Austin TX, 2001.

[56] Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic verification of safety and liveness for xscale-like processor models using WEB refinements. In *Design, Automation and Test in Europe Conference and Exposition, DATE*, 2004. doi:10.1109/DATE.2004.1268844.

[57] Panagiotis Manolios and Sudarshan K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation and Test in Europe Conference and Exposition, DATE*, 2005. doi:10.1109/DATE.2005.257.

[58] Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic verification of safety and liveness for pipelined machines using WEB refinement. *ACM Trans. Design Autom. Electr. Syst.*, 2008. doi:10.1145/1367045.1367054.

[59] Panagiotis Manolios and Daron Vroon. Algorithms for ordinal arithmetic. In *Conference on Automated Deduction CADE*, 2003. doi:10.1007/978-3-540-45085-6_19.

[60] Panagiotis Manolios and Daron Vroon. Integrating reasoning about ordinal arithmetic into ACL2. In *Formal Methods in Computer-Aided Design FMCAD*, LNCS. Springer–Verlag, 2004. doi:10.1007/978-3-540-30494-4_7.

[61] Panagiotis Manolios and Daron Vroon. Ordinal Arithmetic: Algorithms and Mechanization. *Journal of Automated Reasoning*, 2005. doi:10.1007/s10817-005-9023-9.

[62] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Computer Aided Verification CAV*, 2006. doi:10.1007/11817963_36.

[63] Pete Manolios. Reasoning about programs, 2023. URL https://www.ccs.neu.edu/home/pete/courses/Logic-and-Computation/2023-Fall/lectures.html. Lecture notes, Northeastern University, CS 2800: Logic and Computation, Accessed 21 Apr 2025.

[64] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, 1971. URL http://ijcai.org/Proceedings/71/Papers/044.pdf.

[65] Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990. doi:10.1016/B978-0-444-88074-1.50024-X.

[66] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*, 1999. doi:10.1145/313451.313525.

[67] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2016. doi:10.1145/2908080.2908118.

[68] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, 1981. doi:10.1007/BFB0017309.

[69] Amir Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, 1985.

[70] ppopth and libp2p contributors. Gossipsub v2.0 spec: Lower or zero duplicates by lazy mesh propagation. https://github.com/libp2p/specs/pull/653, 2024. GitHub Pull Request #653, libp2p/specs.

[71] ProbeLab. Ethereum network topology. https://probelab.io/ethereum/topology/, 2024. Accessed: 2025-01-15.

[72] Protocol Labs. GO-LIBP2P-PUBSUB. https://github.com/libp2p/go-libp2p-pubsub, 2020.

[73] Protocol Labs. What is Publish/Subscribe. https://docs.libp2p.io/concepts/pubsub/overview/, 2023. Accessed 12 May 2023.

[74] Protocol Labs Research. Gossipsub-v1.1 evaluation report. Technical report, Protocol Labs, 2020. URL https://research.protocol.ai/publications/gossipsub-v1.1-evaluation-report/. Protocol Labs Research Publication.

[75] Protocol Labs Research. Gossipsub: An attack-resilient messaging-layer protocol for public blockchains. https://research.protocol.ai/blog/2020/gossipsub-an-attack-resilient-messaging-layer-protocol-for-public-blockchains/, 2020. Protocol Labs Research Blog.

[76] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 1990. doi:10.1145/98163.98167.

[77] Kumar Shivam, Vishnu Paladugu, and Yanhong A. Liu. Specification and runtime checking of derecho, A protocol for fast replication for cloud services. In *Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems, ApPLIED*, 2023. doi:10.1145/3584684.3597275.

[78] Yu-Chee Tseng, Sze-Yao Ni, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. *Wireless Networks*, 2002. doi:10.1023/A:1013763825347.

[79] Max von Hippel. Code walk of the gossipsub acl2s formalization. https://github.com/maxvonhippel/gossipSub-FM/blob/main/model/demo.lisp, 2023. Accessed 30 May 2023.

[80] Dimitris Vyzovitis. Gossipsub v1.0: An extensible baseline pubsub protocol. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0-old.md, 2020. Accessed 23 Jan 2025.

[81] Dimitris Vyzovitis. gossipsub: An extensible baseline pubsub protocol. https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md, 2022. Accessed 28 Nov 2022.

[82] Dimitris Vyzovitis, Yusef Nasirifard, Yiannis Psaras, and David Dias. Gossipsub: A secure pubsub protocol for unstructured, decentralized p2p overlays. Technical report, Protocol Labs, 2019. URL https://research.protocol.ai/blog/2019/a-new-lab-for-resilient-networks-research/PL-TechRep-gossipsub-v0.1-Dec30.pdf. Protocol Labs Technical Report.

[83] Dimitris Vyzovitis, Yusef Nasirifard, Yiannis Psaras, and David Dias. Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks. *arXiv preprint arXiv:2007.02754*, July 2020. URL https://arxiv.org/abs/2007.02754. arXiv preprint.

[84] Andrew T. Walter and Panagiotis Manolios. ACL2s systems programming. In *Workshop on the ACL2 Theorem Prover and its Applications*, 2022. doi:10.4204/EPTCS.359.12.

[85] Andrew T. Walter, Benjamin Boskin, Seth Cooper, and Panagiotis Manolios. Gamification of loop-invariant discovery from code. In *Proceedings of the Seventh AAAI Conference on Human Computation and Crowdsourcing, HCOMP*, 2019. doi:10.1609/HCOMP.V7I1.5277.

[86] Andrew T. Walter, David A. Greve, and Panagiotis Manolios. Enumerative data types with constraints. In *Formal Methods in Computer-Aided Design, FMCAD*, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_25.

[87] Andrew T. Walter, Ankit Kumar, and Panagiotis Manolios. Proving calculational proofs correct. In *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2023. doi:10.4204/EPTCS.393.11.

[88] Pamela Zave. Using lightweight modeling to understand chord. *Comput. Commun. Rev.*, 2012. doi:10.1145/2185376.2185383.

[89] Chonghe Zhao, Yipeng Zhou, Shengli Zhang, Taotao Wang, Quan Z. Sheng, and Song Guo. DEthna: Accurate Ethereum network topology discovery with marked transactions. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, pages 1711–1720. IEEE, 2024. doi:10.1109/INFOCOM52122.2024.10621281.

Vita

Ankit Kumar was born and raised in Dhanbad, India to Dr. Binay Kumar Singh and Dr. Mamta Kumari. He has an elder sister, Dr. Shilpa Singh. He received a Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology (BHU), Varanasi in 2010. He then worked at Morgan Stanley and Olacabs before pursuing graduate studies. He received a Master of Technology degree in Computer Science from the Indian Institute of Technology Kanpur in 2017. In 2018, he joined Northeastern University to pursue his doctoral studies in Computer Science under the supervision of Professor Panagiotis Manolios. In 2024, he married Yoshita. In January 2026, Ankit joined Amazon Web Services, Inc. as an Applied Scientist.

Permanent Address: Ankit Kumar, son of Dr. B. K. Singh
Buddh Vihar Colony
Dhaiya, P.O. I.S.M.
Dhanbad — 826004
Jharkhand, India

This dissertation was typeset by the author using LaTeX and compiled with `pdflatex`.