Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

Topology-aware protocols, tools and applications for large-scale distributed systems

par

Valerio Schiavoni

Thèse

présentée à la Faculté des Sciences
pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury:

Prof. Pascal Felber, Directeur de thèse
Université de Neuchâtel, Suisse

Dr. Etienne Rivière, Co-Directeur de thèse
Université de Neuchâtel, Suisse

Prof. Peter Kropf
Université de Neuchâtel, Suisse

Prof. Roberto Baldoni
Università di Roma "La Sapienza", Italie

Prof. Ernst Biersack
EURECOM, Sophia-Antipolis, France

Soutenue le 11 Avril 2014

UNIVERSITÉ DE NEUCHÂTEL

Faculté des sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel - Suisse
Tél: + 41 (0)32 718 2100
E-mail: secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Monsieur Valerio SCHIAVONI

Titre:

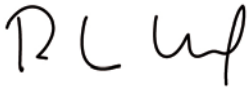
“Topology-aware protocols, tools and applications for large scale distributed systems”

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, Université de Neuchâtel, directeur de thèse
- Prof. Peter Kropf, Université de Neuchâtel
- Dr Etienne Rivière, Université de Neuchâtel
- Prof. Ernst Biersack, EURECOM, Sophia-Antipolis, France
- Prof. Roberto Baldoni, Università di Roma "La Sapienza", Italie

Neuchâtel, le 30 avril 2014

Le Doyen, Prof. P. Kropf



ACKNOWLEDGMENTS

I would like to thank my advisors, Prof. Pascal Felber and Dr. Etienne Rivière, for their great guidance, support and lessons learnt throughout the course of the past four years. They let me join a great research group, and I had the chance to work side-by-side with very smart people. I would like to thank Prof. Baldoni, Prof. Biersack and Prof. Kropf for being part of my jury. I'm very grateful to Prof. Vivien Quèma who, back in 2009, forwarded me the job offer that eventually lead me to Neuchâtel. I would like thank all the brilliant researchers with whom I had the pleasure to co-author papers and to discuss about Computer Science, especially during these years at the Computer Science Department (IIUN) of the University of Neuchâtel. I take the opportunity to thank Miguel Matos and Prof. Rui Oliveira from the University of Minho, Portugal. Our fruitful collaborations made us achieve wonderful results and I hope to continue these collaborations in the years to come. I would like to thank my former colleagues at the INRIA Grenoble, France: they firstly instilled in me the willing to pursue a doctoral thesis in computer science, despite my initial reluctance. I'm grateful for the many not strictly Computer Science-related interesting discussions with my colleagues at the IIUN, especially during the coffee breaks or the table-tennis breaks.

I was lucky enough to be surrounded by old and new friends, who continuously encouraged and supported me. A special thanks goes to Daniele, Luca G., Giulia S., Luca B. and Alessio. Every time I had the opportunity to visit my family in Rome, I was overwhelmed by love, hugs, and food. For these reasons, I would like to send a huge *grazie* to all my family.

Finally, this long journey would have not been the same without the loving support of my Giulia. She accepted my research habits through these years, including the many *white nights* spent in front of my laptop, being encouraging during the hard times, sharing the excitement of a success and supportive when failures occurred: *Thank you!*.

ABSTRACT

Large-scale distributed systems offer scalable solutions to the ever increasing demand of efficient, online services. Examples of such services include data dissemination, group and membership management, distributed indexing and storage, data streaming, etc. The internal mechanisms of these large-scale systems rely on cooperation among thousands of host machines, deployed at geographically distant sites. The cooperation is typically implemented by message-passing (MP). Pragmatically speaking, MP consists is the exchange of sequences of Bytes through physical and logical routing layers. The physical and logical interconnections between the hosts, i.e., their topology, define the routes of the messages. These topologies consistently affect the routing behaviors of the application-level messages. They expose physical properties (i.e., delays, available bandwidth, loss rate, etc.) as well as dynamic characteristics (number of hops, connectivity, contention on the specific link, failure of the end nodes, etc.). The proper design of distributed systems requires taking into account the underlying topologies.

This thesis presents protocols, tools and applications that consider adapting to the routing topology substrate as a key design aspect for large-scale distributed systems.

First, we address the problem of creating anonymous and confidential communication channels on large scale networks. These networks make the design of such confidential communication systems challenging under many perspectives: their scale, the unpredictable crashes of nodes, the inhability to establish direct node-to-node communication channels, etc. We present Whisper, a protocol and its possible applications to establish anonymous and confidential communication channels targeting such challenging network topology conditions.

Then, we observe the need to easily evaluate distributed systems under varying network topology conditions. As a matter of fact, despite the vast literature on the topic, we still lack an integrated tool for topology emulation that is easy-to-use, scalable, featuring multi-user support, concurrent deployments, non-dedicated access, and platform portability. This thesis contributes SplayNet, an integrated tool to support rapid development and evaluation of distributed systems under different network topology conditions.

Finally, this thesis presents Brisa and LayStream, respectively a data-dissemination protocol and a video-streaming application. These two protocols share the common goal of providing reliable dissemination protocols for large-scale networks. Brisa efficiently organizes the nodes to quickly react to failures in the underlying routing topology or nodes. LayStream presents the lesson learnt in supporting a demanding distributed system, such as video streaming, on top of principled composition of gossip protocols.

Keywords: distributed systems, topology, overlay, peer-to-peer, gossip, privacy, anonymity, emulation, user-space, data dissemination, streaming, video streaming, churn

Contents

1 Introduction **1**

1.1 Context 1

1.2 Motivations and Objectives 4

1.3 Background 5

1.3.1 The Splay Integrated Testbed 6

1.3.2 The Peer Sampling Service 6

1.4 Contributions 8

1.5 Outline 9

2 Whisper: Topology-Aware Confidential Communication Protocol for Large-Scale Networks **11**

2.1 Introduction 11

2.2 Problem Definition and Overall Architecture 13

2.2.1 System Model 13

2.2.2 Practical PSS Implementation and NAT Resilience: *Nylon* 13

2.2.3 The Whisper Architecture 16

2.3 WCL: The Whisper Communication Layer 18

2.3.1 WCL Path Construction Algorithm 18

2.3.2 Peer Sampling and Public Key Sampling Services 21

2.4 PPSS: The Private Peer Sampling Service 22

2.4.1 Private Group Management 22

2.4.2 Overlay Management 23

2.4.3 Persistent Paths 23

2.5 Evaluation 24

2.5.1 Experimental Settings . . . 24
2.5.2 Biased Peer Sampling Service . . . 25
2.5.3 Management Cost of Public Keys . . . 25
2.5.4 Availability of Anonymizing Routes under Churn . . . 26
2.5.5 Delays of Anonymizing Routes . . . 27
2.5.6 Bandwidth Consumption vs. Number of Private Groups . . . 30
2.5.7 Application of WHISPER: private T-Chord DHT . . . 30
2.6 Related Work . . . 31
2.7 Summary . . . 32

3 SplayNet: Distributed User-Space Topology Emulation 33
3.1 Introduction . . . 33
3.2 Related Work . . . 35
3.3 The SPLAYNET Architecture . . . 37
3.3.1 Topology Definition and Parsing . . . 37
3.3.2 Resource Allocation and Deployment . . . 38
3.3.3 User-Space Network Emulation . . . 39
3.4 Evaluation . . . 43
3.4.1 Micro-Benchmarks . . . 43
3.4.2 Macro-Benchmarks . . . 46
3.4.3 Concurrent Deployments . . . 50
3.4.4 Scalability . . . 51
3.5 Summary . . . 53

4 Brisa: Lightweight, Efficient, Robust Epidemic Dissemination 55
4.1 Introduction . . . 55
4.2 BRISA . . . 56
4.2.1 Peer Sampling Service Layer . . . 57
4.2.2 Rationale . . . 58
4.2.3 Emergence of a Dissemination Structure . . . 59

4.2.4 Preventing Cycles . . . 60
4.2.5 Parent Selection Strategies . . . 61
4.2.6 Dynamism . . . 62
4.2.7 Generalized Dissemination Structures . . . 63
4.2.8 Multiple Dissemination Structures . . . 64
4.3 Evaluation . . . 65
4.3.1 Structural properties . . . 65
4.3.2 Network properties . . . 67
4.3.3 Robustness . . . 69
4.3.4 Support for multiple trees . . . 70
4.3.5 Comparison with existing approaches . . . 73
4.4 Related Work . . . 77
4.5 Summary . . . 79

5 LayStream: A Layered Approach to Gossip-based Live Streaming 81
5.1 Introduction . . . 81
5.2 Live Streaming Components and Requirements . . . 83
5.3 Gossip-based Building Blocks . . . 84
5.3.1 Peer Sampling . . . 84
5.3.2 Topology Construction . . . 85
5.3.3 Efficient Broadcast: Dissemination Layer . . . 88
5.4 Evaluation . . . 89
5.4.1 Peer Sampling Layer . . . 89
5.4.2 Topology Construction Layer . . . 92
5.4.3 Dissemination Layer and Client Application . . . 93
5.5 Related Work . . . 97
5.6 Summary . . . 98

6 Conclusion 101
6.1 Summary of contributions . . . 101

6.2 Perspectives 102
6.2.1 Confidentiality and anonymity 102
6.2.2 Data dissemination 103
6.2.3 User-space emulation 103
6.2.4 Network topology discovery 104

A Publications **105**

A.1 Posters 106

References **107**

Chapter 1

Introduction

1.1 Context

The rise of large-scale distributed systems

We live in an era of intensive data production and consumption. A recent study from Cisco [1] shows a clear trend in the increasing adoption and growth of web-based services (email, web browsing, online gaming, video streaming, peer-to-peer file-sharing). Figure 1.1 shows how such services shares the total bandwidth consumed across the Internet's backbone. It is obvious how services and applications such as peer-to-peer (P2P) and online video gaming are emerging as the predominant type of network traffic across the Internet routers.

The reasons for this are many. First, the price of computers and electronic equipments felt drastically since the beginning of the 1990's. Figure 1.2 depicts the trends of Producer Price Index (PPI)[1] in the last two decades. Then, there is the occurrence of two concurrent trends: the worldwide costs to connect to the Internet went continuously down [2], and the average global connection speed for Internet residential access increased. The increase in the connection speed to the Internet was first predicted by Nielsen Law's of Internet Bandwidth [3]. The Nielsen Law predicted that the high-end's user connection speed would grow by 50% every year. Figure 1.3 shows how the law predictions have been validated over time. A study [4] carried out by broadband carriers later verified those predictions.

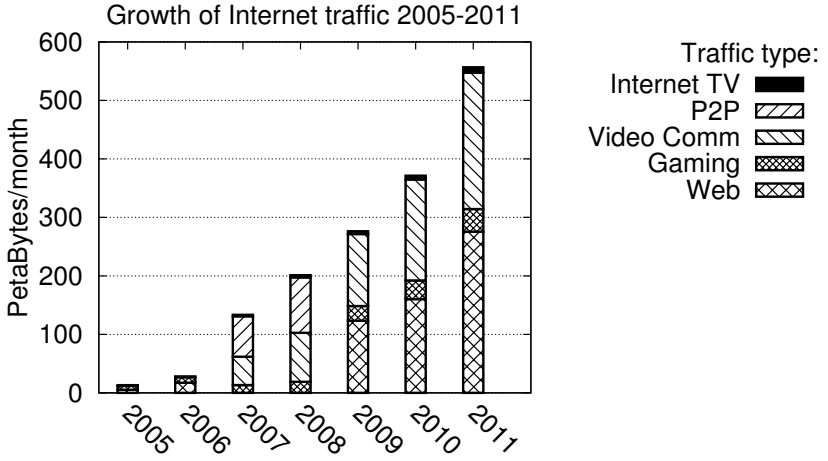


Figure 1.1: Growth of Internet traffic for different online services [1].

[1]The PPI measures the average change over time in the selling prices received by domestic producers.

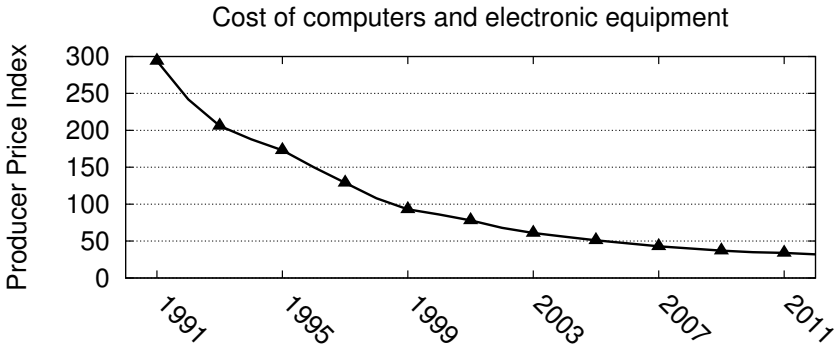


Figure 1.2: Cost of computers and electronic equipment [5].

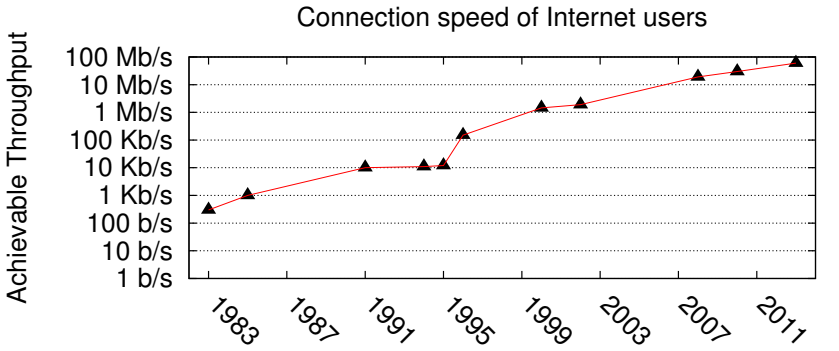


Figure 1.3: Nielsen's Law of Internet Bandwidth.

The combined effects of the decrease in the computer devices costs and the increase of Internet connection speeds for end-users lead to the results depicted in Figure 1.1. Such characterization of the global network traffic makes clear how the most prominent component among the various traffic types stems from P2P protocols' activities. Users rapidly started using P2P applications to address various needs. We can mention few notable examples among popular P2P applications: file-sharing (Gnutella [6], Kazaa [7], BitTorrent [8], Emule [9]), instant messaging (Skype [10]), video streaming (TvAnts [11], PPLive [12]).

All the P2P applications, and the protocols upon which they are built, rely on the basic mechanism of a fully decentralized exchange of messages between end-nodes. Every node acts as a client emitting messages. At the same time, it also acts as a server receiving messages from other nodes, possibly replying back to the client. Be it a chunk from a video stream or block of data from an audio file, P2P protocols, and distributed protocols in general, provide mechanisms to transfers data units across geographically remote machines.

Topology challenges with large-scale distributed systems

The basic node-to-node communication mechanism adopted by distributed systems is named *message passing*. The message passing paradigm allows the exchange of messages (a potentially empty sequence of Bytes) between processes deployed on remote machines. The routing paths traversed by the messages can be analyzed along two perspectives: the physical routing paths, and the application-level routing paths. Such routing paths implicitly define a *topology* of interconnections across the machines, typically modeled via a graph where machines vertices and links used for routs are edges. In this sense, we refer to the *physical-topology* as the set of links and Internet routers that physically connect the machines involved in the distributed

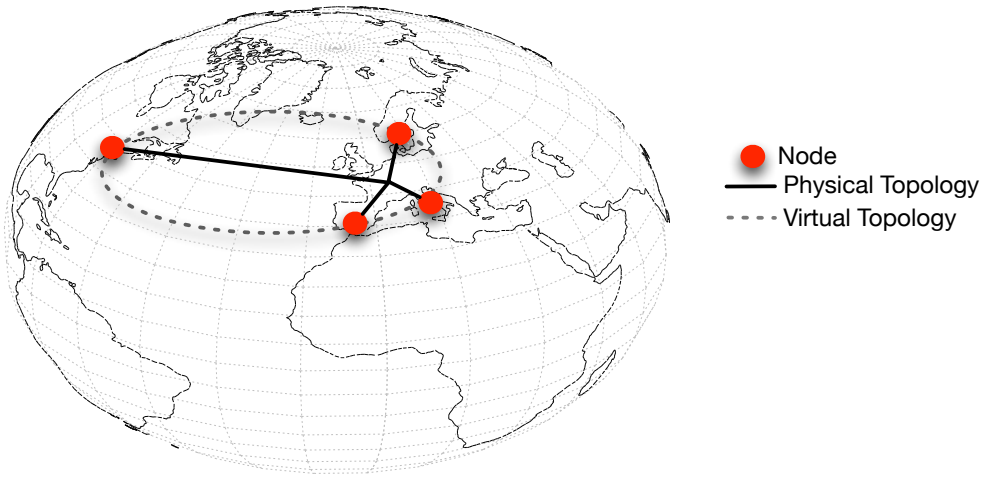


Figure 1.4: Virtual and physical topology: two distinct routing layers in a distributed system deployed over geographically remote sites.

system. The links in the physical topology transport the message bytes from one end to the other. Conversely, we refer to the *virtual-* or *application-topology* as the logical organization of the nodes. A simplified vision of this dualism between virtual and physical topologies is represented in Figure 1.4. In the example, three nodes are deployed in Europe and one node is deployed in the USA. Those nodes are organized along a virtual topology ring, a rather common topology for instance for distributed hash tables (DHT) protocols, such as Chord [13]. The virtual topology is depicted by dashed light grey line, while the physical topology is represented by black thick lines.[2] The two topology layers rarely overlap, although researchers are exploring the possibilities offered by their superposition [14].

In terms of impact on the system performance, the two layers play a fundamental role. The achievable throughput performance of a large-scale data dissemination protocol is constrained by the bandwidth available on the inner and outer links of the physical topology. Likewise, the packet drop rates, the latencies or the routing congestions that can arise at routers along the paths are factors to consider while running a distributed system over a large-scale network. Other corollaries to take into account while running a distributed system are possible obstructive or filtering devices along the routing paths that prevent packet from reaching their designed destination. Examples of such devices are firewalls or Network Address Translators (NAT).

The virtual topology also affects the performance of a distributed system. Protocols must consider many aspects of the virtual topology: the node-to-node message delays as a function of the application-level routing hops, how to balance the load across the nodes involved in the message exchanges or how to prevent bottlenecks to arise due to nodes being excessively loaded. Moreover, precautions must be taken to support the inherent dynamism of the network, especially in the large scale: machines can appear or crash following unpredictable patterns, and fault-resilient mechanisms must be integrated in the design of distributed systems to prevent, as much as possible, the loss of data. The study of large-scale distributed systems must thus address the challenges contributed by the underlying physical and virtual topology layers.

[2]The mapping of the actual physical topology interconnecting geographically distant nodes is an open-research problem itself that is out of the scope of this manuscript. The example given in Figure 1.4 only exemplifies the physical/virtual topology dualism.

1.2 Motivations and Objectives

In the previous section we showed the current trends in the worldwide network traffics. The trends are more and more dominated by large-scale distributed systems, such as P2P applications, Internet-based TV or Internet-based video conferencing systems. The deployment of such complex distributed systems at the global scale introduces many difficult problems. We already mentioned how their design must takes special care of the underlying topologies.

Our research is mainly motivated by the need to design distributed systems that address as much as possible such topology challenges. We focus our attention on the following objectives and motivating scenarios.

Confidential communication. It is no secret that P2P systems are often associated with file-sharing applications used to share copyrighted materials. Due to such popular but illegal usage of these applications, end-users are under the threat of censorship. The Snowden's case about the NSA wire-tapping activities[3] further proves how users should not trust their Internet Service Providers (ISPs). Users running distributed applications to transfer confidential content must be able to do so even in the face of potential deep packet inspection and of the monitoring activity being perpetrated by their ISPs, for instance due to law-enforcement orders. Moreover, in today's common scenario of fast home Internet connections, the traffic originated by the end-users flows through ISP-controlled routers, firewalls and NAT devices. These devices offer an easily reachable vantage point to an ISP willing to inspect the network packets sent and received by the machines exploiting the Internet connection they offer. Although solutions exist to protect the privacy of end users, confidentially and anonymity largely remain an open issue. The first objective of this thesis is to provide a sound solution to solve this problem.

Topology emulation. Network devices such as NATs and routers can be considered as the tip of the iceberg of a broader characterization of realistic Internet topologies. The properties of the physical links interconnecting end-user devices to Internet services, as well as the routers, also play a fundamental role in the achievable performance a distributed system can attain. Certainly they do not come second in terms of impact factor when compared to NATs and routers. Unfortunately, research is often based on simulated, rather simplified topology models, largely ignoring the underlying physical topology. These simplifications lead to results that are far from the reality of concrete systems running on the Internet, drastically reducing the value of the lessons learnt by the evaluation of research prototypes. Techniques such as network emulations are largely adopted to evaluate distributed systems under more realistic conditions. Nevertheless, the current state-of-the-art alternatives can be very difficult to use: they require dedicated, perhaps costly resources, tedious setup and dedicated hardware. They lack of support for multiple concurrent emulated topologies. All these reasons negatively impact the appeal of the existing solutions offered by the academic and open-source communities. The second objective of this thesis is to provide an easy-to-use, scalable and integrated tool to evaluate the performance of a distributed system under varying topology conditions.

Large-scale data dissemination. The last objective of this thesis is to propose efficient data dissemination protocols for large-scale networks. This goal is of paramount importance,

[3] http://www.theguardian.com/world/video/2013/jun/09/nsa-whistleblower-edward-snowden-interview-video.

especially in the light of the trends highlighted in Figure 1.1. Applications such as video streaming will polarize even more the worldwide network traffic in the next years. It is challenging to deploy P2P data disseminations on large-scale networks: nodes fail continuously, messages get lost, routing bottlenecks can prevent a smooth playback of the data stream, etc. We want investigate gossip-based techniques to support such demanding applications. The last part of this thesis is dedicated to implement reliable, lightweight data dissemination protocols that tolerate node faults and that are suitable for the Internet scale and conditions.

1.3 Background

This Section introduces some background concepts and tools upon which the following chapters will build. It is organized as follows.

Section 1.3.1 introduces the SPLAY integrated testbed. Its featured development tools and deployment facilities are adopted throughout entire thesis to facilitate the implementation and the evaluation of the presented distributed systems and applications. Moreover, the SPLAY mechanisms are exploited and extended to realize the network topology emulation features presented in Chapter 3.

Section 1.3.2 presents the basic mechanisms of the peer sampling service framework, one of the most-commonly adopted building block in the context of distributed systems for large-scale networks. These mechanisms will be used as building blocks for the protocols and applications presented in Chapters 2, 4 and 5.

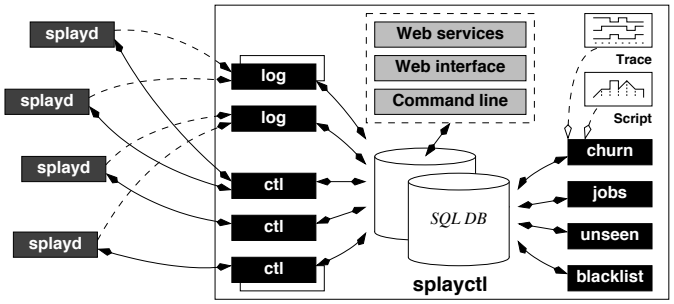


Figure 1.5: Architecture of SPLAY [15].

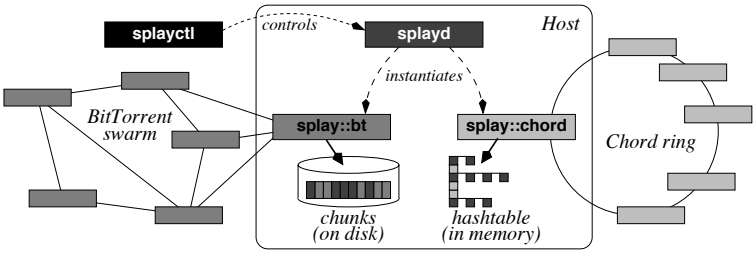


Figure 1.6: Example of the deployment of two applications using SPLAY [15] on a testbed: a BitTorrent [16] file-sharing application, and the Chord [13] distributed hash table.

1.3.1 The Splay Integrated Testbed

The contributions presented in this thesis leverage or extend SPLAY [15], an open-source distributed systems evaluation framework. SPLAY's goal is to ease rapid prototyping and development of distributed protocols. It features a concise and easy-to-learn domain-specific language based on the Lua programming language.[4] The associated libraries support the functionalities that are typically required to implement distributed algorithms.

The language and libraries allow implementations to be comparable in size (i.e., lines of code) to pseudo-code descriptions. This feature is on par with the initial SPLAY objective of making distributed system prototyping and evaluation simple and fast. An example given by the authors of [15] is the Chord DHT [13]. A running implementation uses 58 lines of code, comparable in size with the pseudo-code in the original paper [13]. One of the most prominent features of the Lua language is its straightforward embeddability and its lightweight runtime. SPLAY exploits this feature to allow the re-use of existing (e.g., C-based) code and libraries. SPLAY facilitates the use of a testbed by providing transparent multi-user resource management and deployment support.

The overall architecture of SPLAY is depicted in Figure 1.5. SPLAY runs a set of lightweight C processes, called the SPLAY *daemons* (`splayd`s), on every node of the testbed. These daemons are deployed once by the testbed administrator. The `splayd`s implement sandboxing by controlling and restricting usage to resources on the nodes. This is useful in a non-dedicated environment, such as a network of workstations, to avoid single experiments to consume much of the physical resources of the hosting machines. As single access point, the SPLAY *controller* (`splayctl`), orchestrates the deployment of applications. It is the sole point of access to the system for users, who do not need to have administrative access or user accounts for the machines of the testbed.

Multiple users can access the system concurrently. This is particularly relevant when deploying SPLAY on a shared testbed (such as a university lab) with a restricted number of machines. The concurrent deployment feature allows users to deploy diverse applications to run in co-existence (though in isolation between each other). Figure 1.6 shows a scenario where a BitTorrent file-sharing application and a Chord distributed hash table are deployed on and executed by the same `splayd`s.

The `splayctl` allows users to select nodes for deploying an application according to various criteria (e.g., geographical proximity to each other, load, etc), and dispatches the code to the chosen `splayd`s. The experiment is monitored and managed directly from the `splayctl`, including retrieving ordered logs from all nodes as a single file. The `splayctl` allows fine grain control of the experiments, for instance by replaying a *churn trace* that describes the dynamics of the system and is replayed by each of the `splayd`s participating to the experiment, individually for each user and for each experiment.

1.3.2 The Peer Sampling Service

One of the challenges of large-scale networks is to cope with their dynamicity. A sound solution is given by a *peer sampling* service, deployed as an underlying mechanism for organizing nodes in the network in a fully decentralized and autonomous manner. Peer sampling transparently deals with the complexity of membership management in large dynamic networks, with nodes joining and leaving continuously, and hence simplifies the building of many complex protocol stacks and applications. One can mention application-

[4] `http://www.lua.org`

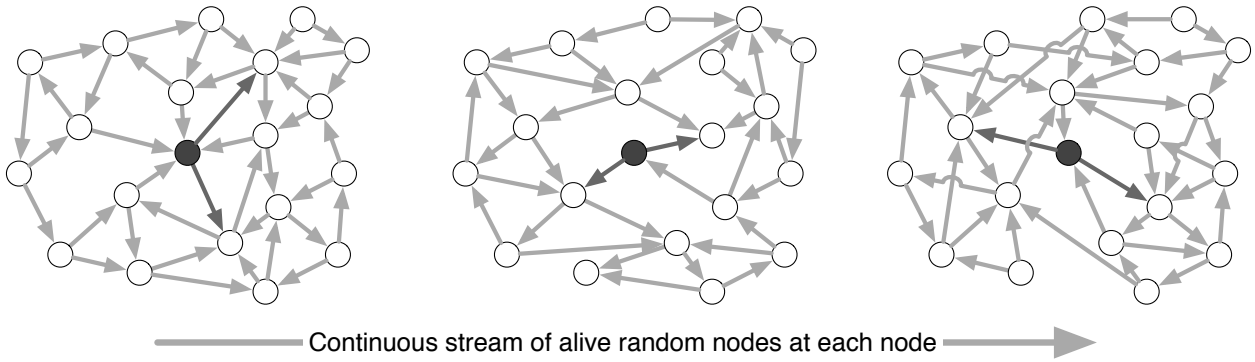


Figure 1.7: Peer Sampling Service: a building block for large-scale applications.

level multicast [17–19], aggregation protocols [20–22], network size estimation [23], overlay construction [24–27], failure detection [28], or network provisioning [29].

A peer sampling protocol, or *peer sampling service* (PSS), provides each node in the system with a continuous stream of alive peers in the system. Typical implementations of a PSS for dynamic systems use gossip-based—or epidemic—protocols [30]. Such protocols are fully decentralized and rely on periodic pair-wise interactions between nodes. Each node has a small limited *view* of the network. For the PSS, this view consists of a set of c other nodes. Periodically, each node contacts another node from its view. Both nodes exchange a subset of their view. Figure 1.7 exemplifies the behavior of the PSS from a node view's standpoint.

Newly received entries are inserted in the local view, which is then truncated to the maximal size c. Such an exchange is typically denoted as a *shuffling* of views [31]. Ideally, the graph built from the local views in a network operating a PSS highly resembles a random graph, with a small diameter, balanced node in- and out-degrees, and strong resilience to disconnection. Membership management is implemented by ensuring that failed or departing nodes disappear from the views of live nodes after a bounded amount of time, and that newly arrived nodes are inserted, and remain, in a sufficient number of other views.

The quality of the overlay created by the PSS is measured by its resemblance to a random graph with fixed out-degrees. A balanced distribution of the nodes' in-degrees ensures load-balancing. A low clustering factor indicates that the diversity of the peers in the views will be maximized: there is no presence of aggregates of nodes that are well-connected together but less linked to the rest of the network. It has been consistently observed by [20–22, 24, 25, 27, 29] that low clustering factors lead to better convergence properties of protocols that use the PSS.

The generic framework to build PSS implementations is presented in Algorithm 1. Several parameters impact on the operation of a gossip-based PSS [30]. In particular, it can be configured along the following three dimensions [30]: *(i)* Gossip target selection: can either be done randomly (rand), or by picking the oldest peer in the view (tail); *(ii)* View propagation: either only the source peer sends its view to the target peer (push), or both source and target peers exchange their view (push/pull); *(iii)* View merging: when truncating a view, randomly chosen peers are kept (rand), or the youngest ones (healer), or the ones received from the other peer (swapper). The framework is organized into two main components or threads: an active thread (lines 1-7), and a passive thread (lines 9-13). The active thread progresses as follows. First, a node selects the target shuffling destination (line 2) from the nodes currently present in the local view. The selection can be based on different strategies, typically based on an age mechanism, or trivially at random. Then, the

Algorithm 1: Generic Peer Sampling Service Framework.

```
1  task every δ time units
2      target ← select_gossip_destination(view)
3      send REQUEST(view) to target
4      if push_pull then
5          RESPONSE(view_t) ← receive(target)
6          view ← merge_and_truncate(view, view_t)
7      increase_view_age()
8  upon receive REQUEST(view_s) from source
9      if push_pull then
10         send RESPONSE(view) to source
11     view ← merge_and_truncate(view, view_s)
```

node executing the active thread sends to the target a shuffle request, embedding its current view in the message (line 3). When the PSS is configured to push/pull mode (line 4), the node waits for a response message from the target. The response embeds the target node's view (line 5). The two views are then merged and, when required, truncated (line 6). Several merging and truncation strategies are possible. Finally, the age associated with each of the entries is incremented (line 7). The passive thread (lines 8-11) follows a symmetric schema, by sending back its local view to the `source` node when configured in push/pull mode, and merging the received view with its local view (line 11).

1.4 Contributions

The main contributions of this thesis are:

A novel network-topology resilient protocol for confidential communications in large-scale networks. The first contribution presented in Chapter 2 of this thesis is a confidential communication protocol targeting large-scale, dynamic networks, where failure of nodes is the norm and direct communication between the nodes is prevented by the presence of NAT devices. We describe the protocols and a supporting architecture. We provide a complete implementation to support an evaluation on a local cluster and a planetary-scale testbed. Preliminary results were initially presented at the poster session of the *9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI) in October 2010, Vancouver, Canada. A full paper [32] was published in the proceedings of the *31th IEEE International Conference on Distributed Computing Systems* (ICDCS), in Minneapolis, Minnesota, USA, June 2011.

An easy-to-use, scalable topology emulation tool. The second contribution of this thesis, introduced in Chapter 3, presents SplayNet, a tool to easily test distributed systems under different network topology conditions. SplayNet extends Splay [15], an integrated open-source framework (introduced in Chapter 1.3) developed at the University of Neuchâtel. We contribute an extensive evaluation and a comparison against state-of-the-art systems. The evaluation demonstrates the soundness of our solution and how it compares favorably

against those. Preliminary results of this work were presented at the poster session of the *17th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS) in March 2012, London, UK. A full paper was later published [33] in the proceedings of the *14th ACM/IFIP/USENIX International Middleware Conference* (Middleware) in December 2013, Beijing, China.

An efficient, robust and scalable data dissemination system. Chapter 4 presents our contribution on the topic of data dissemination for large-scale networks. We contribute a highly efficient, lightweight and fully decentralized data dissemination protocol. The protocol is built on top of a gossip substrate to be highly resilient to faults and is designed to be correct, i.e., to cover all nodes, by construction. We contribute a complete implementation of the system, as well as an extensive evaluation on a cluster environment and on a planetary-scale testbed to compare the design trade-offs and its performance against state-of-the-art alternatives. This work was the result of an international joint effort in collaboration with a team of researchers from the University of Minho, Portugal. This collaboration lead to two main publications. The first paper [34] was published in the proceedings of the *26th IEEE International Parallel & Distributed Processing Symposium* (IPDPS), Shangai, China, May 2012. This article further gained us a **best paper award** in the Application Track of the conference. Successively, we further extended this work, by analyzing more dissemination structures such as forest of trees. This longer version was published as a journal article [35] in the *Journal of Parallel and Distributed Computing* (JPDC).

A study of large-scale gossip-based live streaming systems. The last contribution presented in this thesis reports our experience in building a demanding large-scale system, live video streaming, from the composition of well-principled gossip protocols. We reproduce and validate previous experimental results both from the individual protocols and for the complex system resulting from the composition. Our conclusions lead us to believe that the composition of gossip-based protocols can live up to the expectations when the building blocks are carefully designed. This work stems from a further collaboration with the University of Minho. The results presented in Chapter 5 are currently under evaluation for a conference.

1.5 Outline

This manuscript is organized as follows. Chapter 2 presents Whisper, a middleware system to establish confidential communication channels targeting very large-scale networks, including support for challenging network topology conditions. Chapter 3 introduces SplayNet, an easy-to-use, scalable tool integrated in the Splay framework, to easily evaluate distributed systems under varying topology conditions. Chapter 4 presents Brisa, a lightweight data dissemination protocol that is highly resilient to the underlying dynamic topology, efficiently and promptly supporting node failures. Chapter 5 presents the protocols, the architectural and topological challenges to implement LayStream, a demanding distributed system (a video streaming application) on top of gossip-based protocols. Chapter 6 concludes this thesis and presents some perspectives.

Chapter 2

Whisper: Topology-Aware Confidential Communication Protocol for Large-Scale Networks

This chapter presents Whisper, a confidential communication protocol and its applications in large-scale networks. Whisper provides the support to establish confidential channels between remote machines deployed on today's Internet, where direct communication is typically prevented by the presence of devices such as NAT or firewalls. In this perspective, the physical network topology presents unique challenges to the upper-layer applications. Whisper provides a sound solution to this problem. Moreover, the in-depth evaluation of the Whisper prototype, also supported by a NAT-emulation software layer, indicates and motivates even further the needs for easy-to-use and scalable network topology emulation solutions. These needs will be further motivated and addressed in Chapter 3.

2.1 Introduction

Context

The use of confidential communication between a group of distributed entities is at the core of many applications. Examples include private chat rooms in social networks, information sharing systems guaranteeing freedom of speech, control-flow and admission-control for pay-per-view live-streaming, distributed content indexes that should not be made public to prevent attacks, etc.

The confidentiality of the messages exchanged between the members of a *private group* is typically achieved by the use of encrypted channels that prevent malevolent parties from spying on the exchanged content. However, content encryption alone is insufficient for many of the applications mentioned above, that also require that the composition of the group additionally remain secret, i.e., one should not be able to determine whether or not a node belongs to a group. Furthermore, the existence of the group itself shall also be hidden to unauthorized parties.

Computer networks typically use centralized solutions for supporting private group communication, e.g., by relying on dedicated servers. VPNs allow nodes to create private communication channels with encrypted traffic. Group communication can leverage VPNs, for instance as part of a multi-site company infrastructure, with all communications between sites being routed through some VPN gateways.

In large-scale dynamic settings, where sizable populations of nodes are interconnected in self-organizing and often loosely structured overlays, the use of VPN-based solutions (or similar centralized mechanisms) is inadequate for implementing private group communications. The VPN gateways act as single points of failure, hereby forfeiting one of the major benefits of decentralized systems: their robustness to targeted attacks such as denial of service attacks. As soon as the attackers know the gateway, it is straightforward to disrupt communications within the private group by concentrating the attacks on the gateway. Further disadvantages of solutions based on VPNs or dedicated servers include their scalability to large amounts of users, their price, and their operating costs.

We should point out that these approaches do not keep the membership of private groups hidden from malevolent nodes; only the content of exchanged messages is protected. The process of hiding the communication partners, and hence the identity of the members of the group, must be provided by anonymizing systems such as TOR [36]. These systems rely on a set of dedicated servers that conceal the source of the message from the destination, typically using onion routing mechanisms [37, 38]. Here again, the cost of provisioning such a system with sufficiently many dedicated servers can be a hindrance in large-scale self-organizing networks.

Objectives

These observations make the case for a fully *decentralized, autonomous, and self-organizing* service to support *confidential communications* within groups of nodes in large-scale systems. Unlike existing approaches, this service should let private groups be created by ordinary nodes and emerge within the network, without relying on dedicated and trusted third parties servers. At the same time, it must hide communications between the members of a private group from the other nodes (*content privacy*), as well as keep the group memberships secret to external observers (*membership privacy*). This latter point is especially important, as it is impossible for an attacker to focus an attack on the members of a group without being able to determine their identity.

We target large-scale, Internet-wide networked systems in which a large majority of nodes [39] reside behind network address translation (NAT) devices or firewalls. Our algorithms exploit *peer sampling* as an underlying approach for organizing nodes in the network in a fully decentralized and autonomous manner. We consider *malevolent* nodes that spy upon other nodes in the system, but that follow the protocol specification and do not exhibit other byzantine behavior.

Contributions

This chapter present Whisper, a fully decentralized approach to confidential group communication in large-scale systems in which the traffic may have to take multi-hops paths to circumvent network limitations such as NAT and firewalls. Whisper supports the creation of confidential communication routes without the need for a trusted third party. It additionally provides membership management and overlay maintenance among private groups of nodes communicating in a confidential manner. Our comprehensive evaluation of Whisper in real-world settings indicates that the price of confidentiality remains reasonable in terms of network load and processing costs.

Outline

The remainder of this chapter is organized as follows. We first provide background information on the concepts underlying the WHISPER design in Section 2.2: the peer sampling service, its gossip-based implementation, and the impact of NAT-aware implementations on the design of confidential group communication mechanisms. Sections 2.3 and 2.4 describe the WHISPER components and algorithms. We present in Section 2.5 evaluation results of the WHISPER protocol stack implementation deployed on a cluster and on the PlanetLab testbed. We survey related work in Section 2.6, and conclude in Section 2.7.

2.2 Problem Definition and Overall Architecture

In this section, we first present our system model. Then we provide a detailed overview of the practical aspects related to the implementation of a *peer sampling service*, focusing our attention on its impact on the proposal of private group communications for large-scale systems.

2.2.1 System Model

We consider large-scale networked systems with no centralized entity or trusted third party. We target real-world networks with limited connectivity, rather than idealized settings where all nodes would be able to directly communicate with one another. In particular, we assume that, as observed on the Internet [39], a large majority of nodes reside behind NAT devices or firewalls and can only be reached by using dedicated NAT-traversal mechanisms such as hole-punching or relays. Finally, our system model takes into account churn: nodes are expected to join and leave the network continuously.

We consider the following threat model. Nodes always follow the protocol specification and do not forge, modify, replay, or drop messages. Nevertheless, nodes can be *malevolent* in the sense that they may spy upon other nodes that belong to groups they do not belong to themselves. Confidentiality can be broken in two ways: by observing the content of the messages exchanged between two members of a private group (content privacy), and by determining which nodes belong to a private group (membership privacy). Unless explicitly indicated otherwise, we assume that nodes are not colluding in trying to break confidentiality, i.e., each malevolent node acts on its own.

We consider the links between nodes to be unsafe, i.e., an attacker may be able to observe the traffic sent over a given link. We do assume, however, that the attacker can only control a limited number of links. In particular, it is not able to spy on all the links of a multi-hops path (see Section 2.3) between two nodes.

2.2.2 Practical PSS Implementation and NAT Resilience: *Nylon*

Gossip-based PSS protocols (see Chapter 1.3) typically assume that direct communication is possible between any pair of nodes in the system. However, this assumption does not hold in a real system, where a majority of nodes lie behind NAT devices.[1] For instance, a large study on 7M nodes [39] observed that more than 60% of the nodes were behind NATs. We

[1] A NAT device allows sharing a single connection between multiple peers. It keeps track of existing connections with external peers and translates external addresses into private addresses assigned to the nodes behind the NAT device. All other traffic is typically dropped.

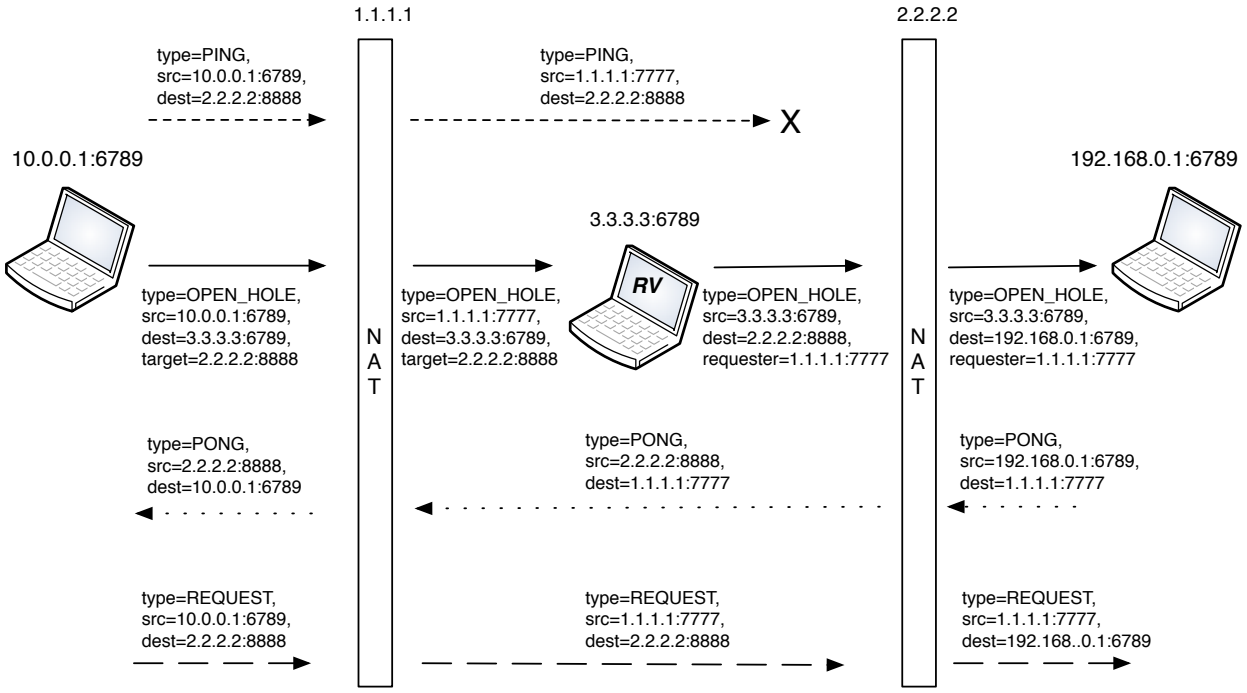


Figure 2.1: The hole punching NAT traversal technique.

refer to the nodes behind a NAT device as N-nodes (*Natted*-nodes), while others are called P-nodes (*Public*-nodes).

Connecting to N-nodes requires implementing NAT-traversal techniques. A widely used approach is *hole punching*. Figure 2.1 depicts this technique. Hole punching requires that the communication is first established by N-nodes with a public *rendezvous* (RV) node, hence opening a connection through the NAT device. The RV node can then share with each N-node the address and port used by the other for this session so that they can communicate directly with each other.

While this method is usable with some NAT combinations [40], there are important cases where hole punching is not applicable (as many as 80% [41]) and the RV node has to act as a relay for all content sent to the N-node. Note that we consider firewalls as being similar to NAT devices w.r.t. to our problem, as their traversal is based on the same techniques.

Deploying a PSS onto a network without considering the impossibility of directly joining N-nodes results in a high imbalance of in-degrees and major negative impact on robustness and connectivity. Therefore, we rely on the mechanisms implemented by the *Nylon* [42] practical NAT-resilient PSS implementation. The remainder of this section details the internal mechanisms and features of *Nylon*, before entering into the specificities of WHISPER. However, we refer the readers to [42] for a thorough evaluation of *Nylon*.[2]

Nylon is a gossip-based PSS protocol that maintains the following invariant: for any node B that lies in the view of a node A, there exists a possibility, known to *Nylon*, to open a communication channel from A to B. This communication channel can be constructed with the help of a chain of P- and N-nodes acting as RVs. Note that the ability of A to communicate with B once the connection has been opened typically lasts longer than the time of presence of the node in the view, and depends on the lease time of association rules

[2]Work on *Nylon* done by the author during his previous role as Research Engineer at the INRIA-Rhole Alpes, Grenoble, France.

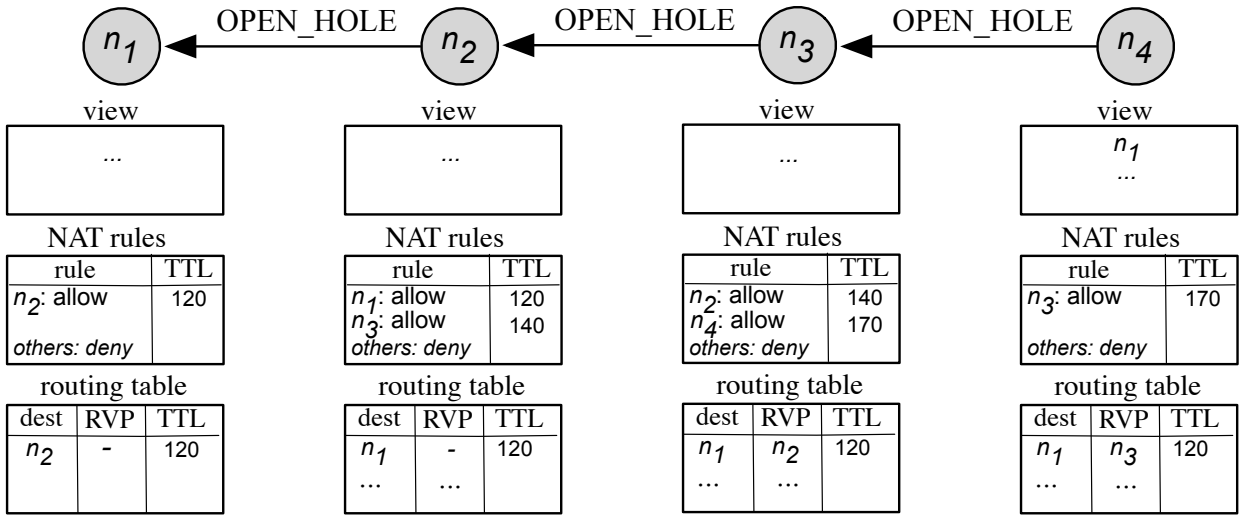


Figure 2.2: *Nylon* operating principle.

maintained by NAT devices. Typical lease times are in the order of minutes for UDP and hours for TCP. Cisco's lease times specifications [43] are of 5 minutes for UDP and 24 hours for TCP. The operating principles of *Nylon* are illustrated in Figure 2.2.

In *Nylon*, hole punching is implemented by mean of a chain of RVs that forward the (protocol-specific) OPEN_HOLE message until it reaches the gossip target. The chain of RVs is built as follows. Consider the case of a N-node n_1 shuffling with a N-node n_2. After having performed hole punching towards n_2 (using a chain of RVs), peers n_1 and n_2 can directly communicate with each other. Thus, they both become RV for each other. Consider now that later, one of them, say n_2, shuffles with the N-node n_3 and gives it a reference to n_1. Before shuffling, peer n_2 performs hole punching towards n_3. Consequently, as between n_1 and n_2, peers n_2 and n_3 both become RV for each other. Finally, consider that n_3 shuffles with the N-node n_4 and gives it a reference to n_1. A chain of RVs has thus been created, as shown in Figure 2.2. This chain allows n_4 to shuffle with peer n_1. For this purpose, it performs hole punching towards peer n_1 by sending an OPEN_HOLE message to n_3 that will forward it to n_2, that will forward it to n_1.

As illustrated in Figure 2.2, in addition to its view, each peer maintains a routing table. This routing table maintains the mapping between a natted peer in its view and its associated RV. For each peer n in the routing table, the RV is the peer it shuffled with to obtain the reference to n. RVs in *Nylon* are constantly changing and following the reactive flavor of the protocol. Also, RVs do not proactively refresh holes (i.e., to keep the corollary rules active by sending messages). Therefore, a time to live (TTL) is associated to each RV entry in the routing tables. TTLs are exchanged by peers together with their views and are updated every shuffle period, and every time a message from one RV stored in the routing table is received.

Observations over a large variety of NAT devices [41] indicate that about 80% of the NAT do not support hole punching mechanisms for TCP, and about 65% for UDP. This results in an important aspect of *Nylon* in the context of WHISPER: in many cases, RV nodes will need to be used as relays for the actual messages exchanged between peers and not only for the connection establishment messages. The mechanism of message relying implemented in *Nylon* is depicted in Figure 2.3.

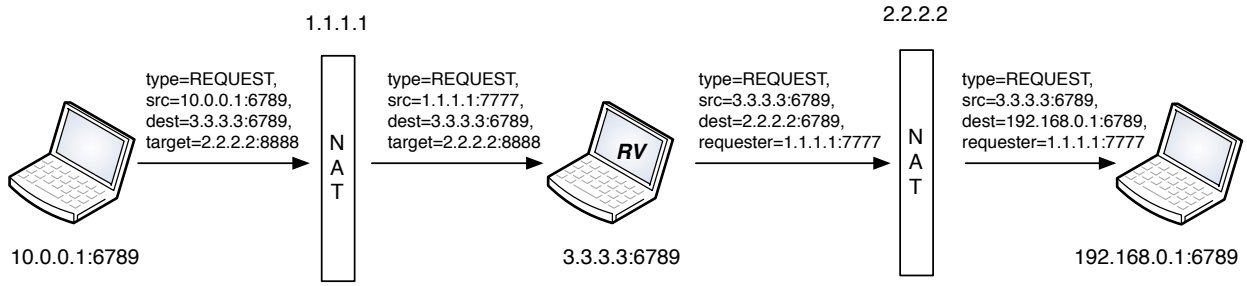


Figure 2.3: Message relaying.

The complete *Nylon* protocol pseudocode is presented in Algorithm 2. The protocol is built on top of the peer sampling framework presented in Section 1.3.2. The only additions to the protocol are for handling NAT traversal techniques and implementing the RV chaining mechanism. The functions to manage the routing tables are not included in the pseudo-code. Instead, these functions are abstracted by four methods. The `next_RV()` function returns the next RV to be used for a given destination. Note that if the destination is directly reachable (because either the destination is public or the peer acts as an RV for the destination), the method returns the destination itself. The `update_next_RV()` method is used to update (or create) an entry in the routing table. It is called whenever a message is received. The `update_routing_table()` method is called to update the routing table. It takes as parameter a view that has been received during a shuffle. This method adds an entry in the routing table for each entry in the view and specifies that the RV for these entries is the peer with which the shuffle was performed. The `decrease_routing_table_ttls()` method is used to decrease the TTL of routing table entries, and purge the expired ones.

Confidentiality implications of *Nylon*

Building private groups directly on top of the PSS would contradict the objective of hiding the content of messages exchanged (irrespectively of the use of *Nylon* mechanisms), as we consider that an external attacker may observe any link. Using a simple encryption mechanism would solve this issue, but the existence of secret communication between two nodes, and hence their participation to a common private group would be exposed and allow group membership mapping. When using the *Nylon* PSS implementation, the content must potentially go through relay nodes that can obtain the same information about the source and destination. The Whisper protocol that we describe in the remaining of this chapter goes beyond this simple but ineffective solutions by allowing the confidentiality of both communications and memberships.

2.2.3 The Whisper Architecture

Whisper is the combination of two layers.

- The Whisper communication layer (WCL) operates on top of the *Nylon* NAT-resilient PSS. It allows confidential communications between peers by protecting both exchanged content and relationship anonymity, even when relays have to be employed for bypassing NAT limitations and individual links may be monitored by an attacker.

Algorithm 2: The *Nylon* protocol.

```
1  task every δ time units
2      target ← select_gossip_destination(view)
3      if (target_NAT = PUB) OR (next_RV(target) = target) then
4          send REQUEST(view, self, target) to target
5      else if (target_NAT = SYM) AND (self_NAT = PRC)) OR (self_NAT = SYM) then
6          send REQUEST(view, self, target) to next_RV(target);          // relaying
7      else
           // Hole punching
8          send OPEN_HOLE(self, target) to next_RV(target)
9          if self_NAT ≠ PUB then
10             send PING to target
11     decrease_routing_table_ttls()
12 upon receive REQUEST(view_s, src, dest) from p
13     update_next_RV(p, hole_timeout)
14     if dest ≠ self then
15         send REQUEST(view_s, src, dest) to next_RV(dest);             // forwarding
16     else if (src_NAT = SYM) AND (self_NAT = PUB)
17     OR (self_NAT = SYM) AND (src_NAT ≠ PUB) then
18         send RESPONSE(view, src) to next_RV(src);                     // relaying
19     else
20         send RESPONSE(view, src) to next_RV(src);          // direct connection
21         view ← merge_and_truncate(view, view_s)
22         update_routing_table(view)
23 upon receive RESPONSE(view_t, dest) from p
24     update_next_RV(p, hole_timeout)
25     if dest ≠ self then
26         send RESPONSE(view_t, dest) to next_RV(dest);                 // forwarding
27     else
28         view ← merge_and_truncate(view, view_s)
29         update_routing_table(view)
30 upon receive OPEN_HOLE(src, dest) from p
31     update_next_RV(p, hole_timeout)
32     if dest = self then
33         send PONG to dest;                          // open the nat hole for dest
34     else
35         send OPEN_HOLE (src, dest) to next_RV(dest);      // forward to next hop
36 upon receive PING from p
37     update_next_RV(p, hole_timeout)
38     send PONG to p
39 upon receive PONG from p
40     update_next_RV(p, hole_timeout)
41     send REQUEST (view, self, p) to p
```

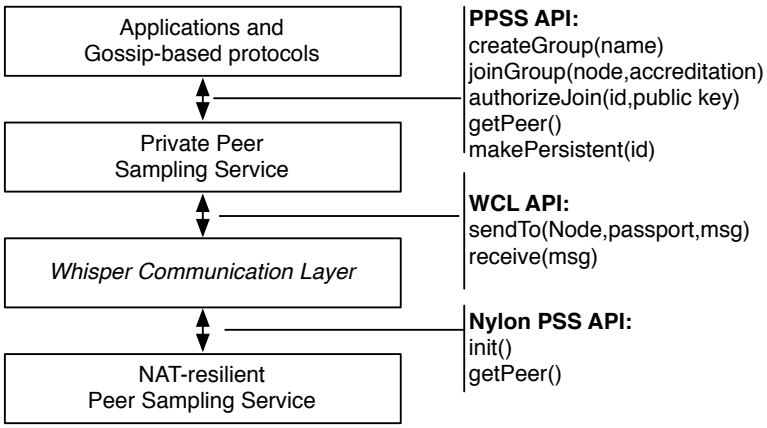


Figure 2.4: WHISPER layers and interfaces.

- The private peer sampling service (PPSS) operates on top of the WCL. It provides the services of a PSS: it acts as a provider of a *private view* of live peers for the applications operating in a private group. It leverages the WCL to guarantee that communications with any node in the private view will remain strictly confidential. The PPSS also deals with group management and membership authentication, and ensures that confidential connections between peers can be maintained even when the destination node does no longer belong to the view of the source node.

Figure 2.4 presents the relations between the various elements composing WHISPER. We describe the WCL and the required modifications to the PSS in Section 2.3. The PPSS will be presented in Section 2.4.

2.3 WCL: The Whisper Communication Layer

The objective of the WCL is to allow two nodes, a source S and a destination D, to communicate in a confidential manner without the need for a trusted third party. Confidentiality here is twofold. (1) *Content confidentiality:* the content of the message exchanged should be visible only to S and D and no other nodes, including the relays that might be used by the *Nylon* PSS layer for bypassing NATs. (2) *Relationship anonymity:* the identity of S and D taken together must not be known by any other node, that is, it is not possible for a node to tell that S and D are engaged in an exchange. This latter guarantee is necessary to ensure that the membership of nodes to private groups can be kept secret to third-party nodes by the above PPSS layer.

2.3.1 WCL Path Construction Algorithm

The WCL provides a one-way multi-hop communication channel between two peers S and D. The content is first encoded by S using symmetric encryption with a random key k (we use AES in our prototype). This ensures content confidentiality.

To guarantee relationship anonymity, we communicate via a path of nodes following the principle of the real-time variant of Chaum's mixes [38], called onion routing [37]. An example is given by Figure 2.5. Onion routing is a technique for anonymizing communications in a distributed system using a path of relay nodes (called *mixes*). Each mix has a private

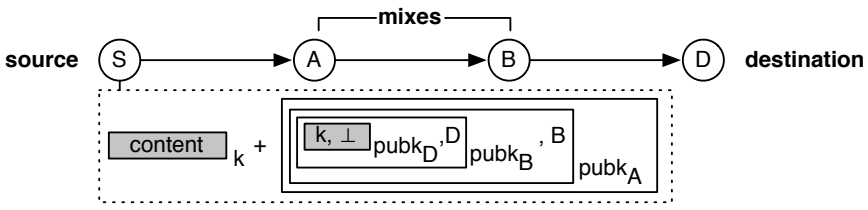


Figure 2.5: A path using two mixes between S and D and the onion-encrypted message generated by S.

and public key pair, and we consider that S, the node preparing the path, knows both the identity of nodes that can act as mixes and each of their public keys (we describe how we support public key management in Section 2.3.2). S encrypts a pair (k,$\perp$) with the public key of D in a variable O. Thereafter, S considers each mix M along the path, in reverse order starting from D, and encrypts O and the identity of the next hop using the public key of M, thus producing a new O. This implies that the decryption of the final O (the *onion path*) needs to be performed in sequence by each of the mixes, using their respective private keys, for determining at each step the identity of the next hop. The last hop (D) learns that it is the destination after decrypting the message, because the identity of the next hop is $\perp$, but this fact cannot be known by the previous mix.

Paths composed of exactly four nodes, including S and D, are sufficient to protect relationship anonymity. Consider a path $S \rightarrow A \rightarrow B \rightarrow D$ with A and B acting as mixes. As A and B have no mean to detect whether the next-to-next hop is $\perp$, they cannot determine whether they are forwarding the message to another mix or to the destination node. For the same reason, neither A nor B can determine whether the node that forwarded them the encrypted message is a mix or the source. Nodes are not colluding in our model so the identity of A's predecessor is not sent along with the message towards B [3]. This guarantees that no intermediate nodes are able to know the identity of A and B simultaneously, thus ensuring relationship anonymity. It also holds when the attacker can obtain the message exchanged on one hop (we consider that the attacker may be able to control one such hop but that it is not able to observe all three links on the path, as it would require a massive control of the network when hops are chosen randomly).

The path from S to D needs to be a valid path, that is, communication must be possible for the three hops $S \rightarrow A$, $A \rightarrow B$, and $B \rightarrow D$. In particular, we need to make sure that NAT-resilient routes are opened. The links $S \rightarrow A$ and $B \rightarrow D$ can leverage the existing NAT-resilient routes that have been opened by the *Nylon* PSS, and for which hole-punching or the setup of relays is still valid. *Nylon* opens such links *on demand* when a node in the view is contacted, either as part of the PSS operation or for the application. Note that thereafter the usability of these opened connections does not depend on the presence of the corresponding node in the *Nylon* PSS view. As detailed in Section 2.2.2, NAT association rules lease times typically range from minutes for UDP to hours for TCP. This is much larger than the typical cycle time of the PSS operation, which is in the order of dozens of seconds [43]. Once a connection has been opened towards a node, the NAT traversal path remains usable as long as association rules remain active.

[3]Note that we can support colluding nodes by simply increasing the length of the path: using f mixes allow to support $f-1$ colluding nodes. We make the assumption that nodes are not colluding to simplify the presentation of the protocols and as it is unlikely to be the case in practice.

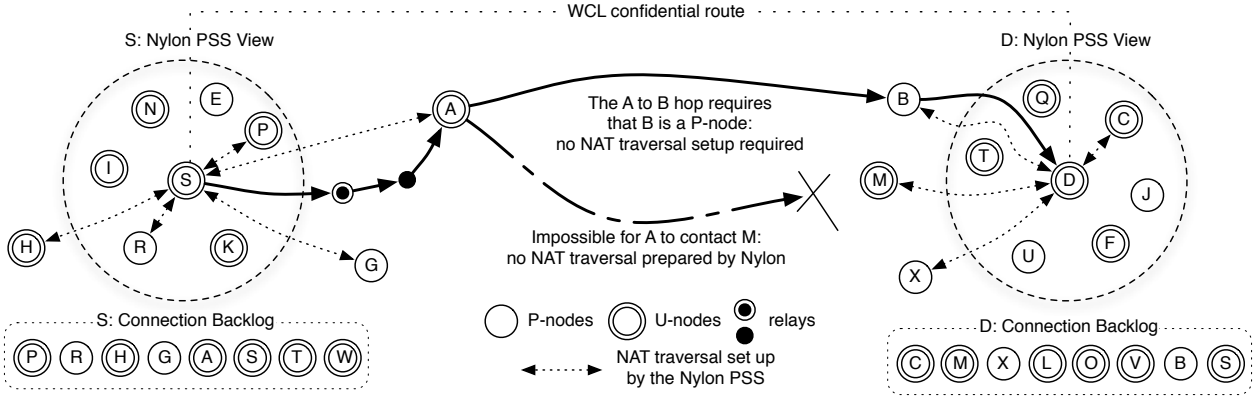


Figure 2.6: Illustration of a WCL path constructed from the CBs of the source and destination nodes. Only nodes that are or have recently been in the CB can be used for constructing the path, and the next-to-last hop as the additional constraint that it must also link to a P-node.

The creation of the routes and the associated state at each node are illustrated by Figure 2.6. Each node builds a *connection backlog* (CB) containing the connection information from $\mathcal{N}ylon$ about the valid hops towards nodes that have been recently contacted. We only consider for inclusion in the CB the nodes that are contacted (or that contact the local node) for a successful gossip exchange, and ignore connections created for the application. The rationale is that gossip exchanges are necessarily bidirectional while this may not be the case for application-level traffic. This means that if node X can contact another node Y through a NAT-resilient route, there also exists such a route from Y to X. The CB is a FIFO queue with maximal size of $2 \times c$ (twice the view size of the PSS). Gossip partners to which a path is opened are inserted at the beginning of the queue and supernumerary elements are removed at its tail. Every cycle, a node contacts a partner in its view and receives, on average, one gossip request from another node. This means that on average two valid new paths will be available during each PSS cycle. Considering for example a PSS cycle time of 10 seconds, $c = 10$ elements in the view, and a backlog of $2 \times c = 20$ elements in the CB, each node will remain at most for 100 seconds in the CB, i.e., much less than the minimal lease time for the NAT association rules.

The CB is used to decide on the $S \rightarrow A$ and $B \rightarrow D$ hops (we consider for now that the CB of D is known to S and will explain how this is achieved in Section 2.4). However the case of the $A \rightarrow B$ hop is different: even if the node S knew the CB of A and could make an informed choice for the next node B, there is no guarantee that there exists a valid node belonging to the CBs of A *and* D with which both nodes can communicate. Therefore, in order to ensure that A can contact B, B must be a P-node that can be reached without restriction by A. This is illustrated by Figure 2.6, where the connection from A to B is possible as B is a P-node, but where the choice of node M as the next-to-last hop would not work: there is no pre-opened path that traverses the potential NATs between A and M. Since the majority of nodes in the network lie behind NATs, we need to maintain a minimal number of P-nodes, denoted by Π, in the CB.

Upon inserting a new node in the CB, we verify that there at least Π P-nodes remain in the queue. If that is not the case, we take a P-node P from the PSS view that does not already belong to the CB, send it an empty message to ensure that a valid path exists from P to the current node, and finally insert P in the CB. This process is repeated as long as there are fewer than Π nodes in the CB. The worst case occurs when all the Π P-nodes of

the CB lie at its tail. In this case, the new node inserted in the CB will result in the removal of the P-node at the tail, which will then be replaced by a new P-node at the head of the CB, flushing out another P-node that needs to be replaced too, and so on. It results that in the worst case, one will need to insert Π new P-nodes from the view. To support this, there must obviously be at least Π P-nodes in the view. To ensure this property, we introduce a modification to the PSS protocol operating with *Nylon*, as described next.

2.3.2 Peer Sampling and Public Key Sampling Services

In order to support the WCL, two modifications are required to the PSS protocol. First, for the connection backlog (CB) to contain Π P-nodes, we need to ensure that there are, at any point in time, at least Π P-nodes in the PSS view. Second, for creating a WCL path, the source node must know the public keys of each mix on the path. The first modification relates to the view truncation after an exchange, while the second complements the PSS with a decentralized public key management service.

P-nodes Availability Enforcement

Upon a view exchange with a partner node, the PSS protocol decides which subset of the view to keep based on one of the following policies (see Section 2.2).

We bias these three selection policies so as to keep a number of at least Π P-nodes in the view. If the original selection policy breaks this condition, we enforce that the Π P-nodes with the smallest age from the view and the received entries are kept even if they would have been discarded by the unbiased policy. Biasing the selection towards P-nodes may lead to an imbalance of the in-degree of P-nodes compared to the N-nodes. A result of this imbalance is that P-nodes will be more loaded than N-nodes. This risk may arise if the percentage of P-nodes that must be kept in the view is higher than the percentage of P-nodes in the network (e.g., if a node requires $\Pi = 3$ P-nodes in a view of size $c = 10$ while only 10% of the nodes are P-nodes). In order to limit the effect of a high value of Π on the load of P-nodes we again bias the selection process and discard in priority the oldest P-nodes that are above the Π threshold.

We will present in Section 2.5 an evaluation of the effect on the quality of the overlay produced by the PSS with and without the P-nodes availability bias. Results show that when Π is set to reasonable values (i.e., a subset of the view size proportional to the ratio of P-nodes vs. N-nodes in the system) the bias has a very small effect on clustering and in-degree balancing.

Public Key Management

In order to create the onion path for the WCL, we need to use the public keys of all nodes on the path (A, B, and D in our example). Therefore, each node must know the public key of the nodes that are in its connection backlog (CB), as well as the public keys of the destination node D and at least one potential next-to-last hop towards a P-node B.

To that end, we build a simple decentralized public key management service on top of the gossip interactions of the PSS. Each time two nodes perform a gossip view exchange, they insert each other in their respective CB. To ensure that nodes know the public key for all entries in their CB, they additionally piggyback their key on the gossip exchange messages. Note that keys are also exchanged with the P-nodes that are explicitly contacted and inserted in the CB when the count of P-nodes falls below Π. This simple mechanism is sufficient for

constructing the first part of the onion path, until the next-to-last hop. S learns the public keys of the last two nodes (D and B) from the gossip interactions over WCL channels, as we explain next.

2.4 PPSS: The Private Peer Sampling Service

The private peer sampling service (PPSS) layer interacts with the WHISPER communication layer (WCL) to implement group-based peer sampling in a strictly confidential manner. The PPSS constructs a *private view* and ensures that a confidential communication is possible with any of its node by leveraging a WCL route. The PPSS is also in charge of maintaining a persistent connection to nodes of the view if required by the application (e.g., as part of a gossip-based overlay construction protocol [24–27]).

The membership of the nodes composing the private group can only be known by the other members of the same group, and the content of the communications between group members cannot be seen by third party nodes, including relays used to bypass NAT limitations. Third party nodes cannot even infer that two communicating nodes belong to the same group. Finally, a node part of several private groups will not disclose its memberships to the members the other groups it belongs to: each group is managed separately by a different instance of the PPSS.

2.4.1 Private Group Management

For joining a private group, a node must first receive an invitation with a temporary signed accreditation and the identity of one entry point in the group. This invitation is initiated by another application running on the system-wide PSS (e.g., in the case of a decentralized VPN emulation), or sent by an external channel such as a Web interface, instant messaging, email, etc.

A private group comprises at least one *group leader* and is associated with a public/private key pair. All nodes in the group know the public key while only one or several leader(s) know the private key. When a new node wants to join a group, it must send its accreditation to one of the leaders. If the accreditation is considered valid (e.g., it can be signed by the group's key or by an external invitation manager), then the contacted leader sends back to the new node its *passport* and the group's public key. A passport is formed by the node's identifier signed with the group's private key. Nodes that belong to a private group ship their passport together with all the communications performed inside the group. When a node receives a message with an invalid passport, the message is simply ignored. This effectively prevents nodes from revealing to non-members their participation to private groups.

The leader(s) selection mechanism is application-dependent. Nonetheless, it is possible to prevent the impossibility of integrating new nodes when the leaders are all offline, by implementing a leader election mechanism based on a gossip-based aggregation [20] of a maximum proposed value: each node in the group that detects that periodic heartbeats sent by leaders are not received anymore can trigger the proposal of a value based on the hash of its identifier. After the aggregation protocol converges (in a few cycles), each node knows the highest proposed value(s) and therefore the new leader(s). These new leaders then propagate a new public key, signed by their identity. This new key is then used along with the previous ones (from a history of public group keys) to verify and issue passports.

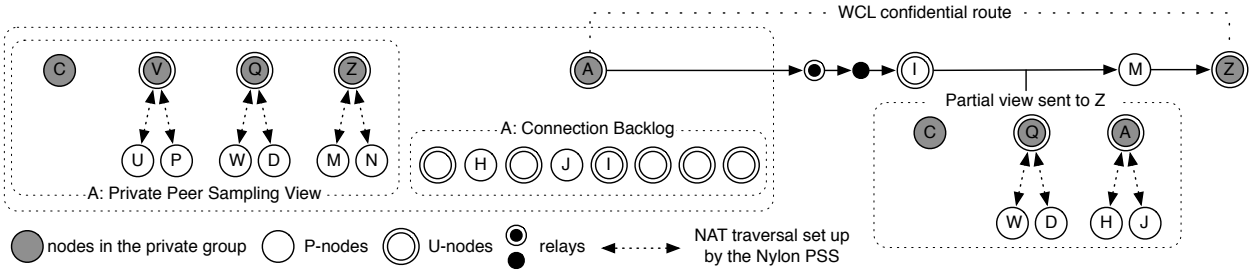


Figure 2.7: WHISPER PPSS state at a node A participating to a private group and creating a WCL confidential path towards a gossip exchange partner node Z based on the information from its private view.

2.4.2 Overlay Management

The PPSS refreshes a node's private view for a given group along the same line as how the PSS refreshes the system-wide views. The exchange of views is based on the same mechanism, but the exchanged entries contain additional information (not just the contact information and age used by the PSS). Indeed, as a node must be able to contact any node in its private view by means of a WCL path in order to proceed to the exchange, it first requires the following information for each node: the identity of the private group partner and its public key. Additionally, for each N-node we need to include a set of Π P-nodes (identities and public keys) that can be used as the next-to-last hop for the WCL path (for P-node we can use any of the P-nodes known, e.g., from the Π P-nodes of the N-nodes in the CB, to use as the next-to-last hop in a WCL route). This additional information is added to the entries of the node's private view. Therefore, upon gossip interaction, the exchanged entries contain both the identities of some private group nodes and the necessary information to contact them in a confidential manner.

Figure 2.7 presents an example of the state kept at a node A participating to a private group. A gossip-based exchange of view is initiated by node A selecting node Z as the partner for the exchange. Both A and Z are N-nodes. Node I in the CB of A is selected for the first step of the onion route. This hop in the path may require the message to transit through one or several relay nodes, depending on the nature of the NAT devices used by nodes A and I. The private view of A for the group includes both the N-node Z and $\Pi = 2$ public nodes that can communicate directly with Z. Node A selects part of its private view (the P-node C and the N-node Q in the example), generates a new entry with its own identity, and sends both components along with its passport to node Z using a WCL path. Node Z, after checking the passport of A, inserts the new elements in its private view and performs the same operation to send its reply to node A.

2.4.3 Persistent Paths

A node can contact the other nodes from a private group as long as they remain in its private view. The gossip-based peer sampling protocol operating on the views does not require connections between nodes to be persistent: only one-time exchanges are necessary to keep the network connected and provide nodes with fresh views of the private group. Nonetheless, applications typically require that some connections be made persistent. For instance, overlay creation protocol such as T-Man [24], GosSkip [25], Kelips [26] or T-Chord [27] maintain a separate view that contains nodes selected based on an application-dependent proximity

criteria. These protocols are oblivious to the fact that the communication with gossip partners takes place using a confidentiality-enforcing mechanism like WHISPER. The only constraint is that they do not try to communicate directly with the other nodes (which would not be possible anyway in most cases due to NAT limitations). Instead, all communications must take place through the WHISPER layer.

A communication channel can be made persistent by means of a simple session mechanism. When a node in the private view is selected by the application as a persistently connected path, this node is kept in a private connection pool (PCP). Each node in the PCP is periodically contacted so as to refresh the Π P-nodes used by the WCL to communicate with it. This periodic refresh can occur at a lower frequency than gossip exchanges as it only depends on the duration of the NAT association rules. This mechanism ensures that communication remains possible and the path is kept persistent transparently to the application.

2.5 Evaluation

We present in this section an evaluation of the WHISPER implementation over a cluster and on the PlanetLab testbed. We evaluate the system in a bottom-up approach, starting with the impact on the PSS of the enforcement of Π P-nodes in the view, the cost of public key management, the availability of WCL routes under varying levels of churn, and the bandwidth and computational costs of the the full WHISPER stack. We finally describe the performance of T-Chord [27], a gossip-constructed version of the Chord DHT [13], operating in a private group on top of the PPSS.

2.5.1 Experimental Settings

Our implementation of WHISPER leverages the SPLAY framework presented in Section 1.3.1. The prototype uses a combination of Lua and C (for computationally intensive operations such as AES and RSA encryption and decryption). In order to support the experiments, we added some NAT emulation features to SPLAY that was not described in the original paper [15]. We note how these initial NAT emulation features are a first step toward more complete network emulation features. The system and tools presented in Chapter 3 will contribute far more complete features to deploy more complex topologies aside with the protocol code. We emulate the 4 major types of NAT devices, (`full_cone`, `restricted_cone`, `port_restricted_cone`, `sym`). Note that `sym` NAT devices require the use of relay nodes by the *Nylon* layer. To reflect real-world scenarios [39], we deployed 70% of the nodes behind NAT devices, evenly split between the four NAT types. Our emulation follows the RFC for TCP-friendly NAT (`http://tools.ietf.org/html/rfc5382`): filtering rules are registered to the emulated NAT devices on a per-connection (for TCP) or per-packet (UDP) basis.

We use the following two testbeds: (1) a local cluster of 22 machines equipped with a 2.2 GHz Core 2 Duo CPU and 2 GB of RAM, connected by a 1 Gbps switched network and supporting up to 1,000 WHISPER nodes; (2) a slice of 400 nodes on the global-scale PlanetLab testbed. For all experiments, we consider a PSS cycle time of 10 seconds and a PPSS cycle time of 1 minute.

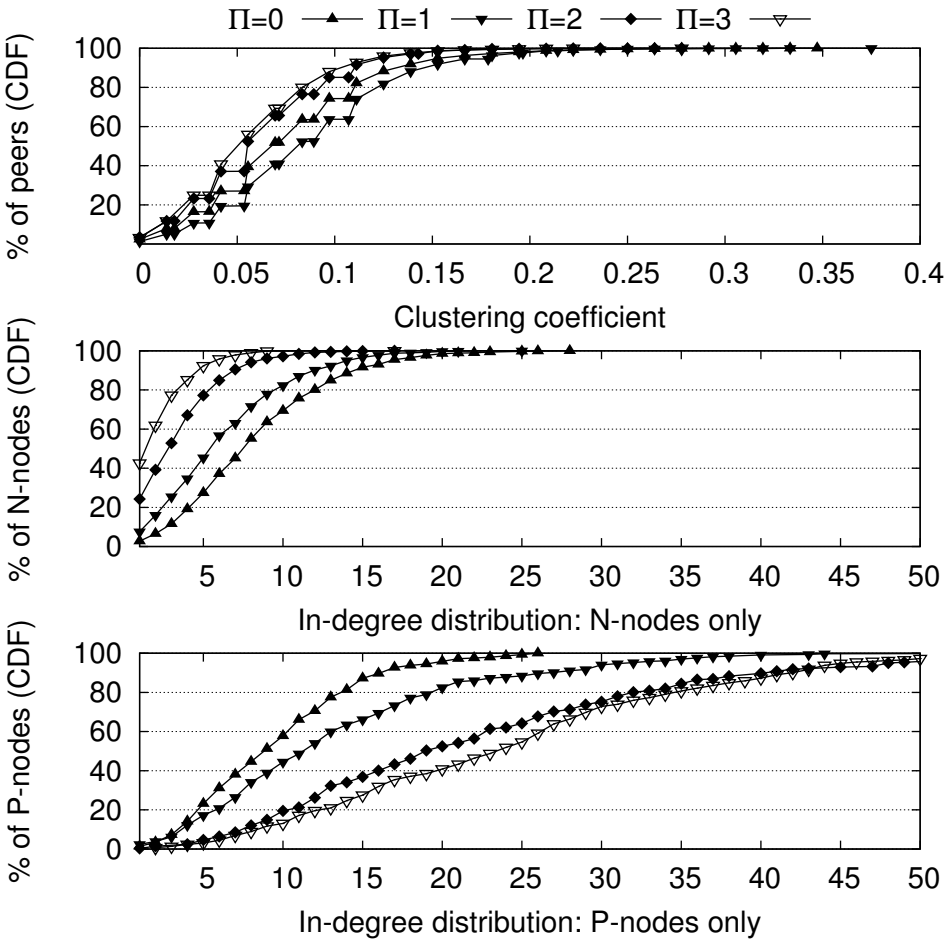


Figure 2.8: Biased PSS: impact on clustering and in-degree distribution.

2.5.2 Biased Peer Sampling Service

We start by evaluating the impact of the modifications to the PSS layer. As detailed in Section 2.3, we modify the view truncation mechanism to ensure that Π P-nodes are kept at each node. We deploy 1,000 nodes on the cluster and experiment with values of $\Pi = 0$ (baseline, unmodified PSS) to $\Pi = 3$. The PSS view size is set to 10 nodes. Figure 2.8 presents the impact of our modification on the two metrics that characterize the resulting PSS graph: the clustering coefficient and the in-degree distribution. We observe that the impact on clustering (upper plot) is negligible. We also compared the distribution of clustering coefficients for N-nodes and P-nodes (not shown on the figure) and found them to be undistinguishable. As expected, the enforcement of Π P-nodes in the views results in a higher in-degree for the P-nodes, as illustrated by the two lower graphs. The choice of Π is indeed a compromise between load imbalance between P- and N-nodes and resilience to churn. In practice, a small value such as $\Pi = 2$ or $\Pi = 3$ is sufficient to allow the creation of WCL routes under high churn conditions, as we demonstrate later in this section.

2.5.3 Management Cost of Public Keys

We operate the public key management on top of the PSS exchanges. Each node initiates one exchange per cycle, henceforth sending and receiving one public key. In addition, each node

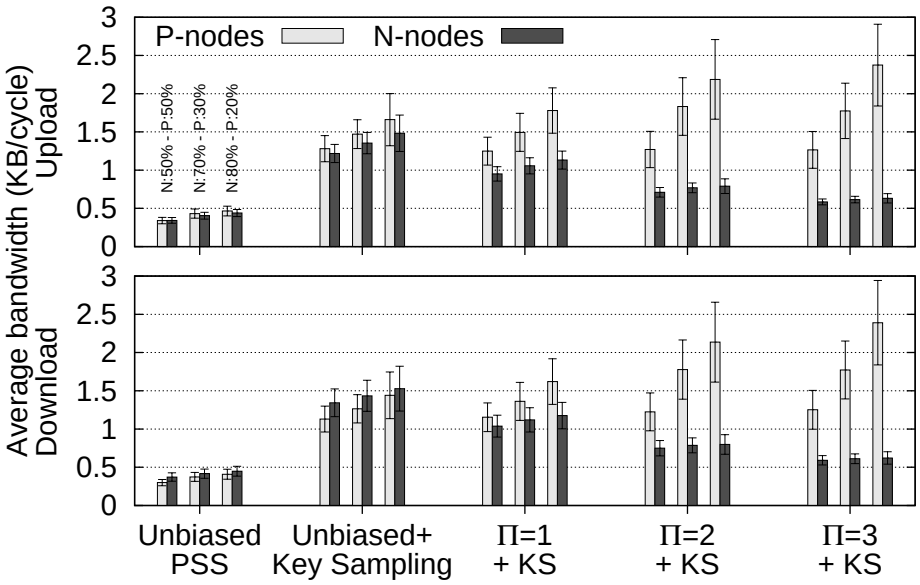


Figure 2.9: Public Key Sampling Service: bandwidth costs (cumulative over 1,000 nodes).

receives on average one request from another node, with the probability of receiving a request depending on its in-degree. As a result of the uneven distribution of in-degrees, P-nodes are likely to have slightly more traffic related to public keys than N-nodes. Figure 2.9 presents the average bandwidth spent per cycle for various configurations: the unbiased PSS corresponds to $\Pi = 0$, with and without key exchanges. As expected, the bandwidth requirements for both types of nodes are balanced. As Π increases, the bandwidth requirement of P-nodes also increases, but they remain within very reasonable margins: a bandwidth of 2.5 KB per cycle corresponds to 256 bytes per second with our PSS cycle period of 10 seconds. We consider three configurations of P- to N-nodes ratios, 50%/50%, our default 70%/30% and 80%/20%. We observe that even in the most extreme case the load of P-nodes for public key management remains reasonable even for $\Pi = 3$.

2.5.4 Availability of Anonymizing Routes under Churn

We now evaluate all WHISPER components together. Our first experiment studies the robustness of the WCL routes in presence of churn. We consider a set of 1,000 nodes (on average), each subscribing to one random group out of a set of 20 private groups, and we set $\Pi = 3$. We use the SPLAY's churn module [15] to inject node arrivals and departures in the system using the script presented at the bottom of Table 2.1. Figure 2.10 graphically display the node dynamism induced by our synthetic churn traces. We vary the amount of dynamism by changing the value X as detailed in the table, from no churn to a high level of churn: the leaving of 10% of the entire network per minute, and the joining of the same amount of new nodes (100% replacement ratio). Creating multiple hops paths in the presence of faults is the more critical aspect of WHISPER. Table 2.1 presents for increasingly-churned runs the following ratios. The ratio of success indicates how many of the WCL paths succeed first-hand, without the need to find an alternative path. We implement the automatic retry and distinguish between two ratios: when such an alternative path exists, and when it does

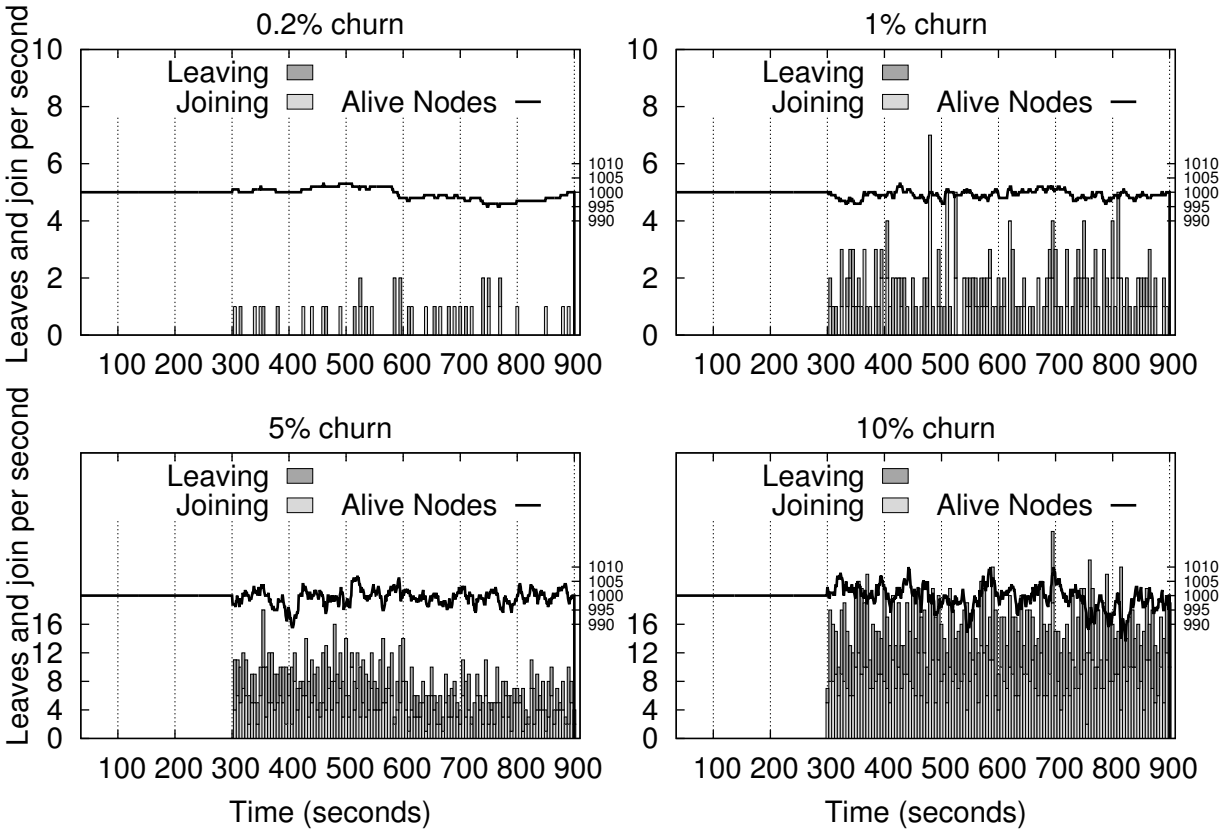


Figure 2.10: Synthetic churn trace, node dynamics (leaves/join). $X = [0.2\%, 1\%, 5\%, 10\%]$.

not.[4] We observe that the success ratio without retry remains extremely high even with high level of churns, and that in a vast majority of cases it is possible to find such an alternative path. This is a result of the periodic shuffling of fresher entries both by the PSS and PPSS protocols. For a churn rate of X=1%, each of the 2.73% of paths that require an alternative, one tries on average 1.0 entry from the BL for the first mix and 1.12 entries from the Π P-nodes of the destination for the second mix to get a correct path (resp. 1.44 and 1.22 for X=5%). This accounts for a large number of successful first retries. In a minority of cases, no such alternative path can be found. This is vastly due to the case where no second mix can be found as the Π P-nodes known to reach the destination all failed. Higher level of alternative impossibility (such as the extreme 10% churn rate) can be mitigated with a larger value of Π (however increasing in-degree imbalance) or a smaller PPSS cycle (at the cost of increased bandwidth).

2.5.5 Delays of Anonymizing Routes

Our next experiment evaluates the costs of operating the WCL routes for view exchanges at the PPSS level. We present the delay breakdown in Figure 2.11 and the average computational loads in Table 2.2.

We observe in Figure 2.11 that the communication latency is largely dominated by network delays. The time required to prepare the onion WCL path, including encryption for

[4]Note that we do not consider that the failure of the destination node is a WCL route failure. Failing to find a path to the destination after Π retries is actually considered by the PPSS layer as a failure of the destination, and the node removed from the private view.

Churn conditions	Success	Alt.	No alt.
No churn	100%	0%	0%
Churn rate: X=0.2% / minute *(30 leaves & 30 joins / 15 min.)*	98.3%	1.42%	0.28%
Churn rate: X=1% / minute *(150 leaves & 150 joins / 15 min.)*	96.7%	2.73%	0.57%
Churn rate: X=5% / minute *(750 leaves & 750 joins / 15 min.)*	96.5%	2.83%	0.67%
Churn rate: X=10% / minute *(1500 leaves & 1500 joins/15 min.)*	90.9%	7.86%	1.24%

```
1 from 0s to 30s join 1000
2 at 300s set replacement ratio to 100%
3 from 300s to 1200s const churn X% each 60s
4 at 1200s stop
```

Table 2.1: Ratio of success on first attempt to construct a WCL route (**Success**), of failed attempts but where an *alternative* route is available (**Alt**) or when no such alternative route exists (**No Alt.**).

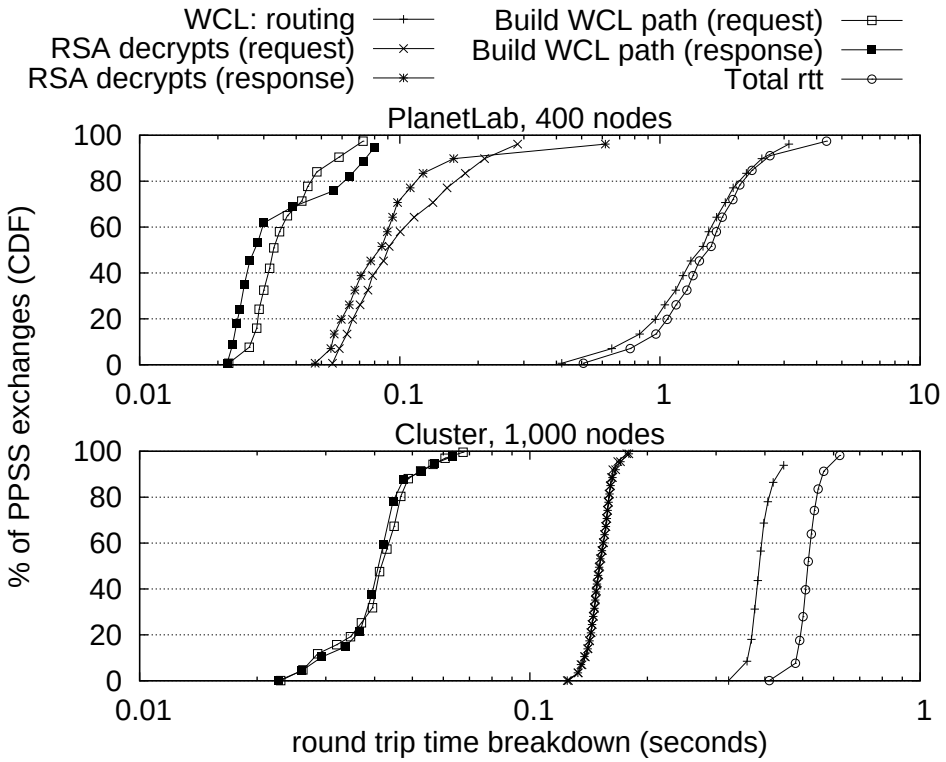


Figure 2.11: Breakdown distribution of the round-trip-times of 1,500 private view exchanges by the PPSS over WCL channels. Exchanged views are at most 20 KB.

each mix, is nearly two orders of magnitudes less than network delays on a cluster as well as on PlanetLab. The decryption of the onion WCL path at each step similarly accounts for a small part of the overall delay. Even on heavily loaded PlanetLab machines with larger network delays and high message loss rates, we manage to confidentially exchange private

	AES	**RSA**	**Total**
N-node	625.9 μs (*< 0.001%*)	293.4 ms (*0.293%*)	294.0 ms (*0.294%*)
P-node	1,506 μs (*0.001%*)	626.1 ms (*0.626%*)	627.6 ms (*0.627%*)

Table 2.2: Average CPU time per PPSS cycle used for encryptions and decryptions with AES and RSA. Percentages indicate the fraction of the 1 minute PPSS cycle a node spends on average for each operation type.

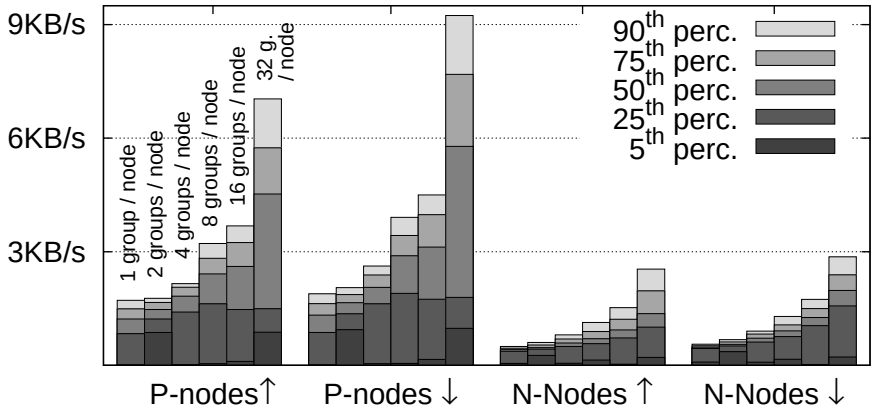


Figure 2.12: Network bandwidth evolution as a function of the number of groups each node participates to (from a set of 120 groups, and for 400 nodes on PlanetLab).

views within 2 seconds for more than 80% of the exchanges. On a cluster, all exchanges take less than 500 ms.

We detail the computational costs associated with encryption and decryption: AES is used to encode the content and RSA is used to encrypt the WCL path components. Table 2.2 presents the average processor time spent by N- and P-nodes during one cycle of the PPSS over a 1,000 nodes network in our cluster (that is, exactly 1,000 exchanges of PPSS views). The size of the exchanged data depends on the ratio of P-nodes in the PPSS view exchanged (as these nodes do not need to be complemented by Π P-nodes as connection means). Each node sends 5 entries of PPSS view. Each N-node entry comes with $\Pi = 3$ P-nodes. Since the size of public keys is 1KB, the maximum size of exchanged subsets of views is thus $\sim$20KB. The computational costs of other operations is completely negligible compared to the cost of encryptions and decryptions, hence we ignore it in the table. As expected, the cost of AES operations (which depends on the size of the message being sent over the WCL path) is very low compared to the cost of RSA (which is independent from that size). We observe that the load on P-node is about 2.13$\times$ higher than for N-nodes, but remains very low: less than 0.65% of one PPSS cycle is spent on the processor for encryption and decryption operations. The reason is that, due to the construction of WCL paths, P-nodes are more likely to act as mixes and hence spend about 4.12$\times$ more CPU time for RSA decryptions.

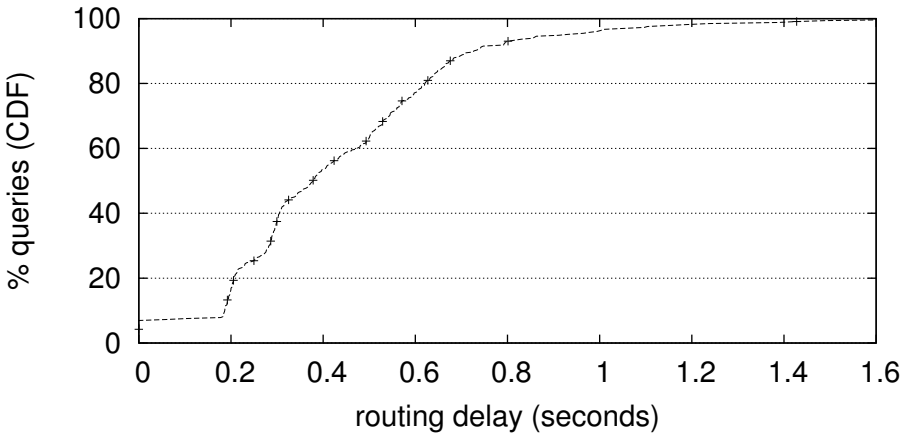


Figure 2.13: Distribution of T-Chord routing delays for a 60-nodes private group among a 400 nodes cluster.

2.5.6 Bandwidth Consumption vs. Number of Private Groups

Our last benchmark of the WHISPER middleware framework evaluates the evolution of the network costs when the number of subscriptions of nodes to private groups increases. We consider 400 nodes on PlanetLab, operating a total of 120 private groups (each P-node creates, and acts as a leader for, one private group). We vary the number of subscribed group per peer from 1 to 32 in a logarithmic fashion. Figure 2.12 presents the evolution of the distribution of upload (↑) and download (↓) bandwidth requirements. We use stacked percentiles with shades of grey to represent a distribution. We observe that, consistently with the values in Table 2.2, the bandwidth costs are more important for P-nodes due to their more important role, yet they remains within reasonable values. As expected, the bandwidth requirements increase linearly with the number of subscribed groups per peer, and the number of views to maintain.

2.5.7 Application of Whisper: private T-Chord DHT

Our final experiment demonstrates the capacity of WHISPER to support applications and higher-level protocols among the members of a private group, so as to transparently benefit from confidentiality and secrecy.

We consider a system of 400 nodes on our cluster. Among them, 60 nodes wish to operate a private index based on a DHT (e.g., to share the location of sensitive data). These nodes bootstrap, within their private group, a Chord DHT overlay [13]. We use the T-Chord [27] gossip-based construction of the Chord ring based on the T-Man overlay construction framework [24]. T-Chord constructs the Chord ring in a self-organizing manner, based on view exchanges with nodes obtained from a PSS (here, from the PPSS private view) and with nodes they are linked to in the Chord ring itself (here, the nodes with which a persistent connection has been opened by WHISPER). Upon view exchange, the nodes select for each type of link in the Chord ring the entries that best match a proximity criteria (e.g., the closest node counterclockwise for the predecessor node) to implement view truncation.

Figure 2.13 presents the routing delays for 350 random queries in the Chord ring. The reply message must route through the WCL layer to protect confidentiality. However, the destination node may not have the querying node in its view. Therefore the querying

node ships its contact information with the query (its identity and public key, as well as any necessary Π P-nodes) so that it can be contacted back by the destination node with a single WCL path. Delays range from 190 ms (smaller delays are for keys held by the querying node) to about 1.5 seconds depending on the length of the routes. Although WHISPER performs extra processing compared to a plain Chord implementation, the lookup latency remains within acceptable limits considering the strong confidentiality guarantees that are provided to the application.

2.6 Related Work

Our contribution relates to work done in the context of anonymizing systems, security and practical implementation aspects of gossip-based protocols, and privacy-preserving gossiping.

Anonymity in networked systems has been first studied in the context of mail anonymization, with Chaum's proposal of using chains of mixes [38], later followed by its real-time variant called onion routing [37]. WHISPER's WCL leverages such onion routes to provide message confidentiality and relationship anonymity in large-scale networks. TOR [36] is a deployed set of anonymizing servers allowing users to browse the Internet anonymously. Cashmere [44] implements each mix using a group of servers for increased robustness. Other examples include AP3 [45], BiFrost [46], SPADS [47], and Tarzan [48]. All these systems consider a set of dedicated servers playing the role of mixes, while WHISPER instead leverages regular nodes in a self-organizing manner and does not require any dedicated infrastructure.

Alternative approaches to the use of mixes include multiple-path routing and network coding mechanisms such as *information slicing* [49], as well as systems based on the dining cryptographers approach such as Herbivore [50] and P5 [51].

Most of research on security aspects of gossip-based protocols have focused on tolerating byzantine faults. Examples include dissemination protocols, e.g., BAR gossip [19] or StarblabIT [52], and byzantine-resilient PSS protocols, e.g., the *secure peer sampling* [53], Brahms [54], Fireflies [55] and PuppetCast [56]. These mechanisms are complementary to WHISPER: we do not consider that nodes act maliciously in operating the protocol but only wish to spy upon other nodes. If we relax this assumption and consider malicious protocol modifications, WHISPER could be combined with one of these solutions to tolerate byzantine faults.

Practical implementations of the PSS for large-scale networks where a majority of nodes are behind NATs or firewalls include *Nylon* [42], upon which we base the design of WHISPER. In [57], Leitao *et al.* propose an alternative approach where nodes do not use NAT traversal to create communication paths but leverage P-nodes as rendez-vous nodes for forwarding gossip exchanges with other nodes.

The *Gossple anonymous social network* [58] proposes to build a decentralized social network, in which nodes with similar interests are linked in a dynamically evolving network, constructed using gossip techniques. It proposes to implement some level of anonymity by means of compact representations of the user profiles and by using gossip-on-behalf: nodes can delegate the management of their view and profile to a third-party node, making the mapping of communicating partners and groups a difficult (but not impossible) task.

Heen [59] presented a decentralized group management protocol that relies on the presence of an underlying Distributed Hash Table to route and disseminate group-management messages. WHISPER builds on top of a gossip-based peer-sampling protocol and dissemination service. Their approach supports a similar threat model, in particular with respect to the

privacy guarantees offered to the upper-layer applications, though it is extended to a Dolev-Yao adversary [60].

The decentralized group-management protocol presented in [61] protects the identity of the members within the same group, whereas WHISPER explicitly allows to identify the membership of a node at least for the nodes within the same group via the passport mechanism.

2.7 Summary

In this chapter we have presented WHISPER, a system that supports confidential communications and private group membership through the use of relationship anonymity in large-scale networks, even when the underlying communication layer is not safe and third party nodes need to be used for bypassing connection limitations such as NAT traversal restrictions. WHISPER is decentralized and self-organizing, and presents a standardized peer sampling interface to other protocols and applications. It does not require any trusted third party node. It allows nodes to create groups and invite other nodes to join, while ensuring that communications between the nodes of the group will not be seen by other nodes, and that the very existence of the group and its members is also kept confidential. We evaluated WHISPER using a deployed prototype and our experiments convey its ability to provide strong confidentiality guarantees with reasonable computational and communication costs.

Chapter 3

SplayNet: Distributed User-Space Topology Emulation

The previous chapter presented a large-scale protocol is resilient to the underlying network topology, in particular when it is characterized by limited/impossible direct connectivity between end nodes. To evaluate those systems under realistic conditions, we extended Splay with libraries to emulate basic NAT behaviors. However, the case of limited connectivity between machines, such as in presence of NAT devices, only represents a first example of the many possible topology conditions a distributed system should consider. Many other factors related to the underlying topology need to be considered while evaluating large-scale distributed systems: the available bandwidth, delay, packet loss rate, contention on the links, etc. This chapter introduces SplayNet, an easy-to-use, scalable, integrated tool to evaluate distributed systems under different network topology conditions.

3.1 Introduction

A key aspect of distributed systems evaluation is the capacity to deterministically reproduce experiments and compare distributed applications in the same deployment context, and in particular when operating under the same network conditions. Distributed testbeds such as PlanetLab [62] allow testing applications in real-world conditions, by aggregating a large number of geographically distant machines. While extremely useful for large-scale systems evaluation, such testbeds cannot be reconfigured to expose a variety of network infrastructures or topologies. Furthermore, the high load and the unpredictable running conditions of shared testbeds are a hindrance for the reproducibility of evaluation results, or for the fair comparison of different applications.

Network emulation supports *controllable* and *reproducible* distributed systems evaluation. It allows running a distributed application on dedicated machines as if it was running on an arbitrary network topology, and observe the behavior of the application in various network conditions. The emulation of communication links is based on an input *topology*, i.e., a graph representation of nodes, routers, and the properties of their connections. A cluster with a high-performance local network can typically support the execution of applications and the emulation of the topologies.

The focus of this chapter is to provide support for easy evaluation of networked applications (e.g., indexing [13], streaming [63], coding [64], data processing over non-standard topologies [65], etc.) under diverse yet reproducible networking conditions. Furthermore, we seek to provide support for concurrent deployments of emulated topologies and distributed applications, where the physical nodes of a cluster can be used for running multiple experiments

with different topologies, without interference and loss of accuracy for any of the experiments. Finally, we posit that the uptake of network emulation mechanisms will be greater if the setup of such mechanisms remain simple and cross-platform, and if they are integrated with a toolkit that facilitates distributed systems prototyping and evaluation, for researchers, students, and engineers. This requires mechanisms and tools for rapid development, deployment, observation, and control of distributed experiments. Note that our work focuses on the evaluation of networked applications on top of standard TCP and UDP connections, when presented with various end-to-end characteristics: bandwidth, delay, packet loss, and congestion. We do not consider the evaluation of the network stack itself, or the evaluation of low-level network characteristics and protocols, which is the focus of others tools [66].

Existing solutions [67–80] support emulation of part or all of the characteristics of a topology, but present a number of limitations. None allows researchers to deploy several network topologies *at the same time* and *on the same physical nodes* over a shared platform. Indeed, they enforce that a node of the testbed is used by one user, for one topology. This requires a large amount of physical resources, or imposes severe restrictions on the number of users and/or the size of their experiments. Furthermore, existing approaches require privileged or *root* access to the machines of the testbed, and often the use of dedicated machines or specialized operating systems to support network emulation. Finally, most of them require to completely reconfigure testbed nodes to emulate a new topology.

Contributions

The main contributions of this chapter are the following:

- We propose a novel approach for supporting network emulation with user-space mechanisms and without support from the operating systems. Our approach allows emulating complex topologies for which existing systems require network queues implemented in the kernel space of dedicated emulation nodes.
- Our approach features configurable and modular network models. It supports complex topologies with inner routers and links, link sharing models, and overheads emulation.
- We introduce a fully decentralized monitoring algorithm for emulation of congestion, delays, and packet loss for inner nodes of the topology, without actually instantiating inner nodes nor requiring a centralized control point.
- We present support mechanisms for network emulation that enable simple selection and sharing of resources between multiple concurrent topologies and application deployments, without need for the user to directly access the physical nodes.
- We describe an implementation of our systems, SplayNet, developed as an extension of the Splay framework presented in Section 1.3.1.
- We evaluate our approach with several micro-benchmarks and networked applications deployed over various topologies. We compare our system to ModelNet [67] and Emulab [68, 69]. Results indicate that SplayNet achieves similar accuracy for network emulation but with lower resource requirements, and supports concurrent deployments without degradation of accuracy. Our approach scales well under heavy load and large topologies, and it can be deployed with minimum management effort.

Name	Mode	Req. HW support	Orchestration	Concurrent deployments	Path congestion	Emulation B	D	P
ModelNet [67]	Kernel-space	×	Centralized	×	√	√	√	√
Emulab [68, 69]	Kernel-space	√	Centralized	×	√	√	√	√
SliceTime [70]	Kernel-space	√	Centralized	×	√	√	√	×
Nist NET [71]	Kernel-space	×	Centralized	×	×	√	√	√
ACIM [72]	Kernel-space	×	Centralized	×	√	√	√	√
P2PLab [73]	Kernel-space	×	Centralized	×	×	√	√	√
IMUNES [74]	Kernel-space	√	Centralized	×	×	√	√	√
Netkit [75]	Kernel-space	×	Centralized	×	√	√	√	√
NetEm [76]	Kernel-space	×	*(not applicable: single link emulation only)*			×	√	√
EmuSocket [80]	User-space	×	*(not applicable: single link emulation only)*			√	√	×
MyP2P-World [78]	User-space	×	Centralized	×	×	√	√	√
WiDS [79]	User-space	×	Centralized	×	×	×	√	√
Mininet [81]	User-space	×	Centralized	×	×	√	√	√
SplayNet	User-space	×	Decentralized	√	√	√	√	√

Table 3.1: Classification of network emulation tools (B/D/P=bandwidth/delay/packet loss emulation).

Outline

The remainder of the chapter is organized as follows. Section 3.2 reviews related work. The design and internals of our system are described in Section 3.3. We present a detailed evaluation of SplayNet in Section 3.4 and conclude in Section 3.5.

3.2 Related Work

We classify work related to SplayNet along several perspectives, presented in Table 3.1. We distinguish solutions based on their operational mode (user-mode or kernel-mode), their need for specialized hardware or dedicated devices (switches, VLANs), and the type of orchestration for the emulation of the traffic at inner nodes/routers of the topology.[1] We also consider the support for *concurrent deployments*: multiple emulated topologies onto the same set of machines, for different users and different applications. We finally consider the ability to emulate traffic congestion along routing paths, as well as end-to-end bandwidth, delay, and packet loss. Although hardware-only emulation systems exist [82, 83], in the remainder of this section we focus on solutions that operate partly or entirely in software. We do not consider emulators specializing in wireless networks [84, 85], nor do we focus on simulation tools [86].

ModelNet [67] uses a set of dedicated machines organized in a cluster, called *emulator nodes*. These nodes are in charge of shaping all the traffic emitted and received by the *edge nodes* supporting the application. ModelNet requires modifying the routing tables of the kernel at edge nodes to redirect all outgoing traffic toward emulator nodes. Traffic shaping rules (bandwidth and delay) are applied to all packets by the means of DummyNet [87] pipes set up in the kernel of emulator nodes. It is possible to deploy only one emulated topology at a time. Every topology modification requires root access to the cluster for redeploying all emulator nodes and updating the kernel routing tables at edge nodes.

[1] *Centralized* means that a single node is in charge of emulating the traffic for a given inner link, while different machines may be in charge of emulating different inner nodes. *Decentralized* on the other hand means that several nodes coordinate for emulating the same inner link.

Emulab [68, 69] is a shared platform that runs experiments on a dedicated emulation testbed. Although Emulab allows users to deploy several experiments under different network conditions, once a machine of the testbed is assigned to an experiment it cannot be used for any other. Emulab uses the same mechanisms as ModelNet [67] to shape traffic. To reduce the number of host machines required by each experiment, Emulab supports an *end-node-traffic-shaping* mode: the application's nodes shape the outgoing traffic themselves, relying on *tc* [76] or DummyNet [87] for, respectively, Linux- and BSD-based experiments.

Some network emulation tools are based on virtual machine deployment utilities. SliceTime [70] solves the time-drifting problem for large-scale experiments by providing a synchronization component to the deployed virtual machines. It relies on the Xen hypervisor [88]. SplayNet does not require the use of a hypervisor on the host machines, it only spawns new user-space processes to accommodate concurrent experiments.

P2PLab [73] relies on DummyNet mechanisms built in a BSD kernel. It organizes emulated networks in subnets. Each physical machine in a P2PLab cluster is responsible for a subnet and manages all the traffic within this subnet. Along the same lines, IMUNES [74] operates through a set of virtual machines interconnected via DummyNet pipes. Its originality resides in the management of the cluster machines hosting the virtual machines, which is driven by a peer-to-peer protocol. The protocol monitors the state of the machines and notifies the other nodes about failures and load conditions. This information is subsequently used when dispatching virtual machines. Network emulation itself operates similarly to other DummyNet-based emulators, and it requires the physical network hosting the experiments to provide programmable VLAN support.

Mininet [81] uses lightweight virtualization mechanisms to emulate software-defined networks on a *single host*. In contrast, SplayNet and the systems presented above target deployments onto a cluster of networked machines, allowing computationally intensive tasks and greater scalability. Other low-level tools aim at shaping the traffic originated by user-space processes. Trickle [89] is a user-space bandwidth shaper for unmodified Unix applications. DelayLine [77] requires the target program to statically link against traffic-shaping libraries. The authors of [80] and [78] both propose user-space emulation tools targeting P2P protocols implemented in Java: the latter provides bytecode-level compatibility with existing applications, whereas the former offers specialized APIs. These systems only support emulation of end-to-end links characteristics and not of complete topologies, thus categorizing them as traffic shapers rather than topology emulators.

In [90], the authors propose to deploy distributed rate limiters (DRL) for general purpose cloud services. Rate limiter nodes synchronize through a lightweight UDP protocol, which shares similarities with our decentralized congestion monitoring approach (Section 3.3.3). DRL does not provide any support for rate-limiting multiple services concurrently running on the same nodes. SplayNet provides a per-destination dedicated token bucket, while DRL mimics the behavior of a centralized token bucket algorithm at each rate limiter node.

The support of concurrent deployments requires appropriate resource selection mechanisms. Since physical network links will be shared by multiple emulated links, the resource selection must ensure that the capacity of the physical link is sufficient for all emulated links. No emulators feature such capabilities, and most require to deploy topologies on distinct sets of nodes, thus greatly impairing scalability. The few systems that support concurrent deployments on the same nodes leave to the user the responsibility of provisioning sufficient physical capacity for emulated links.

The present work represents the first attempt to propose user-space network emulation within an integrated distributed systems evaluation framework. It provides support for

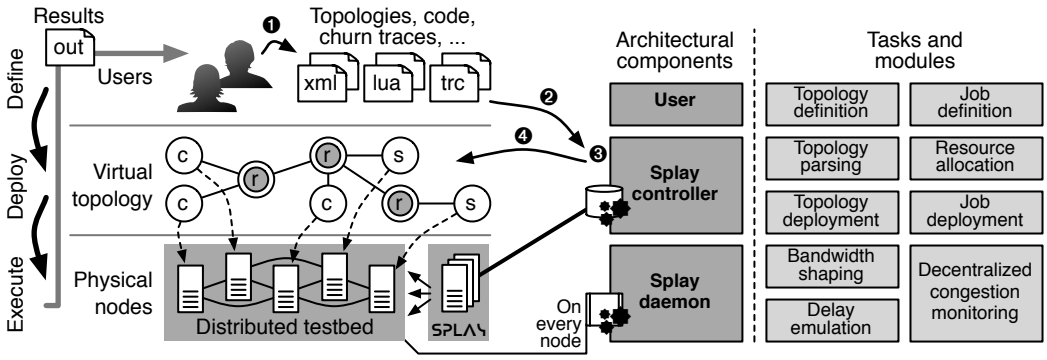


Figure 3.1: The SplayNet architecture.

concurrent deployments while offering comparable performances to single-topology and kernel-space solutions, as will be shown in Section 3.4.

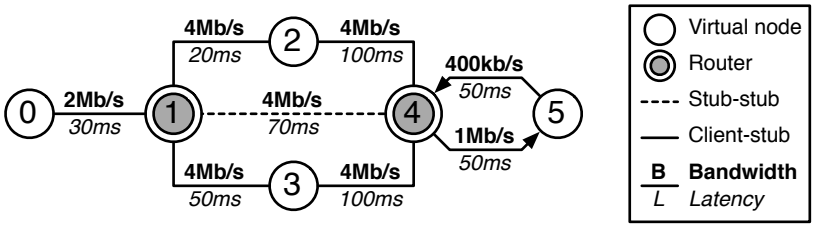
3.3 The SplayNet Architecture

In this section we describe the various components necessary for supporting user-space network emulation and their integration in our SplayNet prototype. Figure 3.1 presents an overview of the architecture.

3.3.1 Topology Definition and Parsing

The first step is to define a network topology to emulate. Users write an abstract description that maps vertices and edges of an undirected cyclic graph to the physical connections of a network (Figure 3.1-❶). Users can specify the interconnections between nodes and routers, as well as the physical properties of the links (delays, bandwidth, and packet loss rate). Application nodes can be inner nodes in the topology (and not only end-nodes), in order to support relay-based applications such as coding [64] or in-network aggregation [65]. SplayNet supports two topology description formats: the ModelNet XML-based language [91] and the Emulab TCL-based language, itself based on the one used by the NS-2 network simulator [92]. A sample topology and an excerpt of its description in XML are given by Figure 3.2. The second step is the deployment (Figure 3.1-❷). The user submits to the Splay controller the topology description, the code to execute, and any additional files required to drive the experiments. SplayNet's topology parser extracts the graph topology. Links in the topology description are uni-directional. Non-connected topologies are rejected. The user can however request implicit link symmetry: when there is no link between two elements but a corresponding reverse link exists, an implicit link can be created, with the same characteristics as the reverse one. This operation does not modify any of the links present in the original topology, thus supporting topologies where both symmetric and asymmetric links coexist. We then use an all-pairs-shortest-path algorithm based on links delays[2] and, for every shortest path, derives the maximum available bandwidth along the path (link with the lowest bandwidth), the overall delay (sum of the delays of individual links), and the packet loss probability (product of the packet loss of individual links).

[2]Upon tie, we select a random link to balance the load but other strategies are possible, e.g., link with minimum latency or maximum bandwidth.




```
(...)
<edge src="1" dest="4" dbl_kbps="4096" delayms="70"/>
<edge src="4" dest="5" dbl_kbps="1024" delayms="50"/>
<edge src="5" dest="4" dbl_kbps="400"  delayms="50"/>
(...)
```

Figure 3.2: Graphical representation of a topology, and an excerpt of its description in XML.

3.3.2 Resource Allocation and Deployment

SplayNet allocates testbed resources for executing the user code on the emulated topology (Figure 3.1-❸). In the context of Splay, this problem corresponds to selecting a minimal set of `splayd`s for executing the job. The allocation procedure ensures that the deployment of a topology does not impair on the accuracy of other deployed topologies, by avoiding saturating the bandwidth of physical links beyond a safety margin. Finding a *minimal* set that satisfies all constraints on a shared infrastructure is a NP-hard problem [93]. Although efficient heuristics are known [93–95], they require knowing the start and duration of all experiments in advance, a requirement that is not met in our context. In SplayNet, we adopt a simple greedy approach to guide the selection of `splayd`s. The objective is that all links in all emulated topologies are supported by physical links with enough available capacity. We do not consider delays as a selection criterion as we assume that SplayNet will be deployed in a cluster where the latencies observed on physical links are stable and much smaller than the latency requested for the emulated paths. The `splayctl` also keeps track of the current load of the machines, as part of the regular Splay operation. The administrator provides the maximal emulated bandwidth that can be emulated on a single physical link. This value depends on the cluster hardware and network. We use a value of 100 Mb/s in our experiments, as illustrated by the concurrent deployment experiment of Section 3.4.3. If several `splayd`s are deployed on the same physical machine, the bandwidth available to each `splayd` is a fraction of the total available bandwidth and this value must be adjusted accordingly. For a new job, we select the least loaded nodes that satisfy the connectivity requirements, i.e., that have physical links to other nodes with sufficient available capacity taking into account the topology being deployed and those already running. If no such set of `splayd`s is found, deployment is not allowed.

We only need to map *application nodes* to `splayd`s. Routers are implicitly emulated by the communication links between the edge nodes. The advantages of this approach are twofold: first, it significantly reduces the amount of resources required to emulate large topologies; second, it frees the system from the burden of having powerful machines dedicated to shaping the traffic at routers. ModelNet [67] adopts a similar technique to reduce the amount of resources required for emulation in its *end-to-end* mode, but it does not emulate congestion at intermediary hops under this execution mode. We emulate traffic congestion at inner nodes with a distributed protocol and a link sharing model, described in Section 3.3.3.

The Splay controller finally dispatches the code to be executed to the selected `splayd`s, along with the topology information required to initialize the network emulation

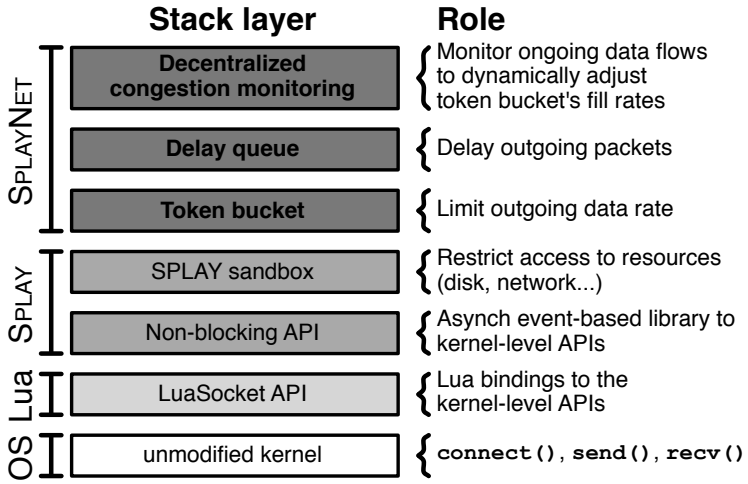


Figure 3.3: User-space network emulation stack.

layer (Figure 3.1-❹). This information is encoded with a compact marshaller that has negligible overhead on the traffic sent to the nodes. As an example, the information necessary to emulate the topology of Figure 3.2 adds only 430 bytes to the data sent to each `splayd` for the job deployment.

3.3.3 User-Space Network Emulation

SplayNet performs link and topology emulation only in user-space, and independently for the different deployed jobs on the same `splayd`. This brings a number of benefits. First, administrators do not need to have privileged access to the machines of the testbed nor to set up any hardware network infrastructure, since the emulated network layers are initialized at the application level. Second, it overcomes a common limitation of most other state-of-the-art systems by supporting the emulation of several topologies simultaneously.

Figure 3.3 depicts the application-level stack of SplayNet and the role of each of its components. In the remainder of this section, we discuss the design and implementation of the layers that perform link delays emulation and user-space bandwidth shaping, as well as the decentralized congestion monitoring protocol and associated model that emulate the effects of congestions at inner nodes. The mechanisms presented below support both TCP and UDP streams through unmodified socket operations.

Latency and Packet Loss Emulation

Links of the topology are first characterized by latency values and packet loss rates. To account for the associated delays, the `splayd` instantiates a *countdown queue* for each outgoing link of the node of the topology being emulated. Outgoing packets traverse this queue before they reach the network. A countdown timer is initialized to the link latency value when a packet enters the queue and, upon expiration, the packet is sent over the wire. Note that the actual latency of the physical topology is assumed to be orders of magnitude smaller than the emulated one, as all `splayd`s are typically executed on a cluster. Otherwise, the value of the countdown timer should be adjusted to take into account delays observed at the physical level.

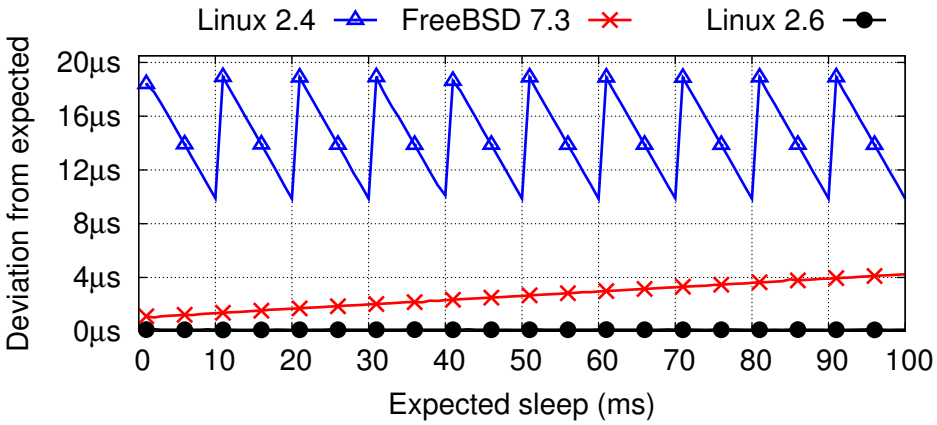


Figure 3.4: Scheduling accuracy.

The reactivity to the timer expiration is crucial for accurate emulation, especially when emulating low-latency links, thus the choice of the underlying operating system plays an important role for achieving good performance in link delay emulation. We evaluated the scheduling accuracy on various operating systems, and reproduced results on par with those presented in [96]. Figure 3.4 presents our evaluation. Scheduling accuracy is around $0.1\mu s$ for Linux 2.6, and in the order of a few μs for Linux 2.4 and FreeBSD 7.3. This indicates that accurate latency emulation is achievable, with measurable errors in the order of a few microseconds.

Packet loss is enforced by simply dropping random packets at the source according to the calculated loss rate on the path to their destination. Here again, we assume that the underlying physical network has a negligible packet loss rate that we do not need to compensate.

Bandwidth Shaping

In addition to latency, a topology specifies the maximal bandwidth for each of its links. The actual bandwidth available to the application will be smaller, and depends on the size of the messages sent through the socket. Our model takes into account emulation of overhead as follows. For TPC/IP and UDP/IP, we use the default Ethernet MTU size of 1500 B (bytes). Ethernet overheads consist of 38 B for each message: 12 B of source and destination addresses, 8 B of preamble, 14 B of header, and 4 B of trailer. We then add the overhead of IPv4 (20 B), and TCP or UDP headers (20 and 28 B, respectively). The overhead factors in the number of packets for a given application-level message, and determines the bandwidth that is actually used on the emulated link. This overhead model, which can be easily modified to account for different network settings, allows us to precisely emulate the actual bandwidth available to an application sending messages of various sizes. It is also independent from the configuration of the supporting physical network (e.g., the use of jumbo frames).

We use a token bucket algorithm [97] to cap the throughput of outgoing traffic to the value specified in the emulated topology.[3] The algorithm operates by inserting a number of tokens at a fixed rate (determined according to the available bandwidth) into a virtual

[3]The *tc* [76] tool integrated in the Linux kernel uses a similar approach to bandwidth shaping. However, SplayNet is cross-platform and does not rely on any kernel support as it integrates its own shaping mechanism.

bucket. Each token represents a fixed amount of bytes that can be sent. Application-level packets are delivered over the wire only if the corresponding amount of tokens is available in the bucket. Otherwise, they are re-queued in the bucket. This simple strategy guarantees a consistent average throughput during emulation.

The bucket fill rates are initially configured to the minimum available bandwidth across all hops on the shortest path between the source and destination nodes. Afterwards, fill rates are dynamically adjusted by the decentralized congestion monitoring protocol based on the actual available bandwidth on the path, dynamically considering other flows taking place in the topology.

Decentralized Congestion Emulation

The delay emulation and bandwidth shaping mechanisms are the foundations of a decentralized network emulation platform, and are the first components of the emulated network model. They are however not sufficient for accurately emulating network congestion across multi-hop routing paths. This task is the responsibility of a decentralized congestion monitoring protocol, which constitutes the second part of our model. Note that the network model is modular: both parts can be modified independently of the emulation framework, and new models can be integrated, with different overheads, link sharing, or QoS policies.

In a centralized solution such as ModelNet [67], one or a small set of dedicated hosts are continuously keeping track of the network traffic on all possible paths of the topology, since all packets are routed through these hosts. This global view of the network allows throttling the data rates according to the limits imposed by the topology.

We advocate the use of a decentralized architecture that does not require specific nodes to handle all traffic passing across the topology. Instead, we rely on a distributed protocol to promptly distribute notifications about the start and end of data streams. These notifications are disseminated to all the nodes involved in the emulation of a given topology through fast and reliable UDP multicast channels (PGM).

View update. Whenever a node starts or stops sending data using TCP, it first updates its local view of ongoing network flows by incrementing the number of competing flows on every hop from itself to the destination node. Then, it disseminates this information to the other nodes by specifying the source, the destination, and the virtual routing hops involved in the stream. In the context of a large-scale topology deployment (Section 3.4.4) with 150 nodes, we observe average dissemination delays of 7.36 ms. Upon receiving this information, the other nodes adjust their local view accordingly by updating the number of competing streams on affected links and, if necessary, the token bucket's fill rates. In the case of UDP streams, it is not possible to determine the end of a communication as with TCP. Hence, we adopt a periodic report strategy: every 50 ms, the amount of data sent through the socket is propagated to other nodes, which update their state based on information from the previous period.

Each node needs to maintain an up-to-date view of ongoing data flows on the emulated network, whether originated by itself or by other nodes, and determine how internal links bandwidth is shared between competing flows. This view is efficiently modeled as a n-ary tree rooted at the local node. The leaves of the tree represent the other virtual end-nodes, while inner nodes correspond to the routers. Edges of the tree are labeled with their maximum bandwidth capacity and latency, and they embed a counter that keeps track of the number of active data streams on the associated links. Each leaf is augmented with a token bucket that specifies the maximum data rate allowed to reach the corresponding node. The initial

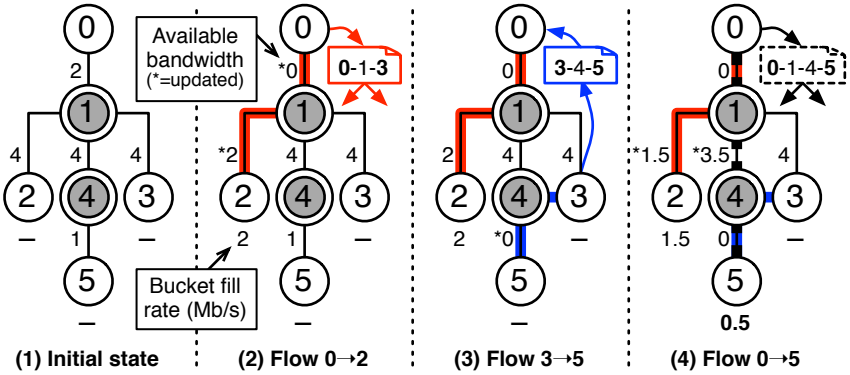


Figure 3.5: Evolution of the tree maintained by node 0 for the topology of Figure 3.2 with the establishment of 3 communication flows.

fill rate for each token bucket corresponds to the bandwidth allowed by the path from the local node to the leaf.

Link sharing model. Whenever multiple streams share a segment of the routing path, the token bucket fill rates are adjusted to split the bandwidth between the competing streams, for each of the internal links of the topology supporting multiple streams. The split depends on a bandwidth sharing models. The basic *Max-Min* sharing model introduced in [98] does not correctly reflect actual sharing behaviors [99]. Therefore, we use the *RTT-aware Max-Min* sharing model [100, 101], which is widely considered as accurate. First, the allocation of bandwidth ρ_i for each flow f_i on a link is capped by the limitation of its bandwidth-delay product: the flow is capped by the ratio of the sending window size W_i and roundtrip RTT_i: $\rho_i \leq W_i/RTT_i$.[4] Second, the sum of ρ_i for all flows on the link must not exceed the capacity of the link F. The share ρ_i of the available bandwidth for each flow is then inversely proportional to the flow RTT_i, i.e., $\rho_i = F \times ((RTT_i)^{-1} \sum_{j=1..n} (RTT_j)^{-1})$ when the first capacity constraint does not apply to any flow. The allocation takes into account the fact that some hops in end-to-end paths are not able to use their full share of a given emulated link. In this case, the remaining bandwidth is redistributed to other existing communication flows under the model constraints until no further refinement is possible.[5]

Example. Figure 3.5 illustrates the tree maintained by node 0 in the topology of Figure 3.2 as communication flows are established between nodes. Initially, no communication takes place and the buckets are idle (first tree in the figure). Then, node 0 starts communicating with node 2. To that end, it sends to other nodes information about the path that has been established, and it updates its local view by adjusting the available bandwidth on links and the fill rate of the bucket at leaf 2 (second tree in the figure). After receiving a message from node 3 that starts sending data to node 5, node 1 simply updates the available bandwidth on the links but does not need to change the bucket fill rates as there is no competition with one of its communication flows (third tree in the figure). Finally, node 1 communicates with node 5. The new flow competes with the previous two as it shares a link with each of them: link 0→1 for the first flow, and link 4→5 for the second. The sharing of these links is determined according to the RTT-aware Min-Max sharing model and the bucket fill rate of leaves 2 and 5 are adjusted accordingly (fourth tree in the figure). From the topology description in Figure 3.2, we obtain the following RTTs: 0→2 is 100 ms, 3→5 is 300 ms

[4]We use the default value of 64 KB for the sending window size W_i.

[5]Note that the current model considers that the reverse-path bandwidth is sufficient to accommodate the traffic of ACKs. Refinement of the model may include these aspects, e.g., based on [102].

and 0→5 is 300 ms. As a result, the 2 Mb/s of the link 0→1 are shared as 75% of 2 Mb/s = 1.5 Mb/s for 0→2, and 25% of 2 Mb/s = 0.5 Mb/s for 0→5. Note that the bandwidth allocated to flow 0→5 is actually the maximal allocatable, as link 4→5 is shared with flow 3→5 with the same RTT for both flows.

3.4 Evaluation

In this section we present an extensive evaluation of our contributions. We compare SplayNet with the *de facto* reference network emulators ModelNet [67] and Emulab [68, 69]. Similarly to SplayNet, both systems provide complete emulation toolsets, from a topology description language to topology deployment facilities. We use the same application code over the three emulation systems, by using Splay and Lua stand-alone libraries on ModelNet and Emulab.

In Section 3.4.1 we first present a set of micro-benchmarks that measure the accuracy of the delay and bandwidth emulation on simple yet representative topologies. Our study then proceeds with a set of macro-benchmarks based on real-world applications (Section 3.4.2). We use the Chord DHT [13] as an example of delay-sensitive application, collaborative application-level multicast using parallel n-ary trees [63] as an example of a bandwidth-sensitive application, and network coding [64] as an example of a topology-sensitive application. One of the distinctive features of SplayNet is the support for concurrent deployments of multiple topologies on the same testbed. In Section 3.4.3, we investigate the scalability and accuracy of SplayNet when concurrently deploying several topologies, and we illustrate the need for taking into account physical network resources when selecting deployment nodes. Finally, Section 3.4.4 concludes this evaluation by presenting the behavior of SplayNet when emulating large and complex topologies.

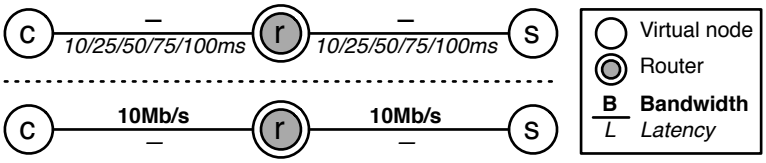
We setup a SplayNet cluster on top of a 1 Gb/s switched network with 60 machines, each with 8-Core Xeon CPUs and 8 GB of RAM. The ModelNet cluster is deployed on the same machines. We used the similarly powerful `pc3000`[6] machines for Emulab experiments.

The SplayNet modules executed by the `splayd`s for network shaping are implemented in pure Lua. We use version 5.1.4 of the Lua virtual machine for all the experiments. The `splayctl` extensions are implemented in Ruby. Due to the small number of machines typically available on Emulab, we had to restrict our evaluations on this platform to a maximum of 20 nodes per experiment.

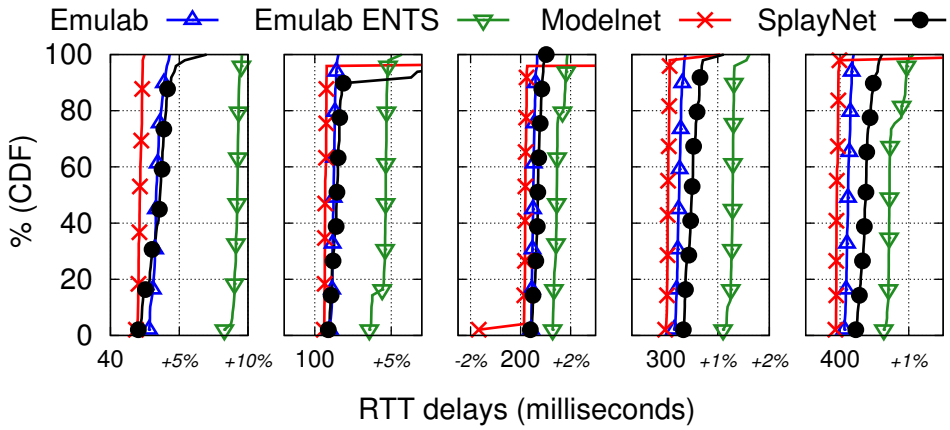
3.4.1 Micro-Benchmarks

Latency. To evaluate the accuracy of link latency emulation, we deploy a simple client-server application using remote procedure calls (RPCs) at the edges of the topology, as shown in Figure 3.6.a (top). We measure the accuracy of the RPC's round-trip-time (RTT) for increasing emulated latencies. This experiment also includes results for Emulab configured in end-node-traffic-shaping (ENTS) mode to remove any latency overhead toward a third-party shaping node. Figure 3.6.b presents the cumulative distribution function (CDF) of observed delays. The expected RTT is shown on the y-axis for each of the link latency values, with variations expressed as percentages. Performance over the 3 testbeds is very similar: emulated latencies never deviate more than 10% from the expected values, and never more than 5 milliseconds in absolute terms.

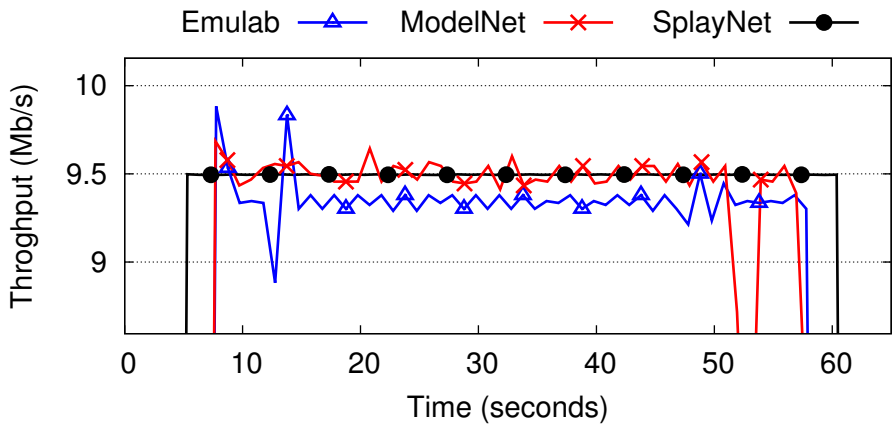
[6] `http://emulab.net/shownodetype.php3?node_type=pc3000`



(a) Topologies for latency (top) and bandwidth (bottom).



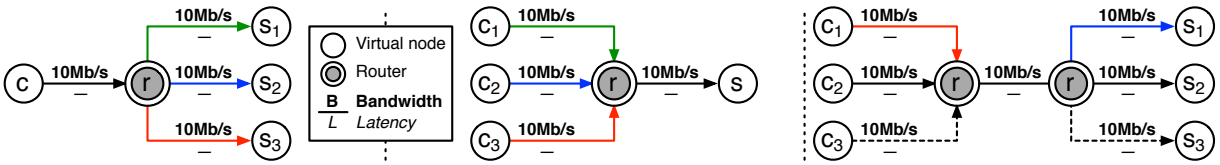
(b) Link latency emulation.



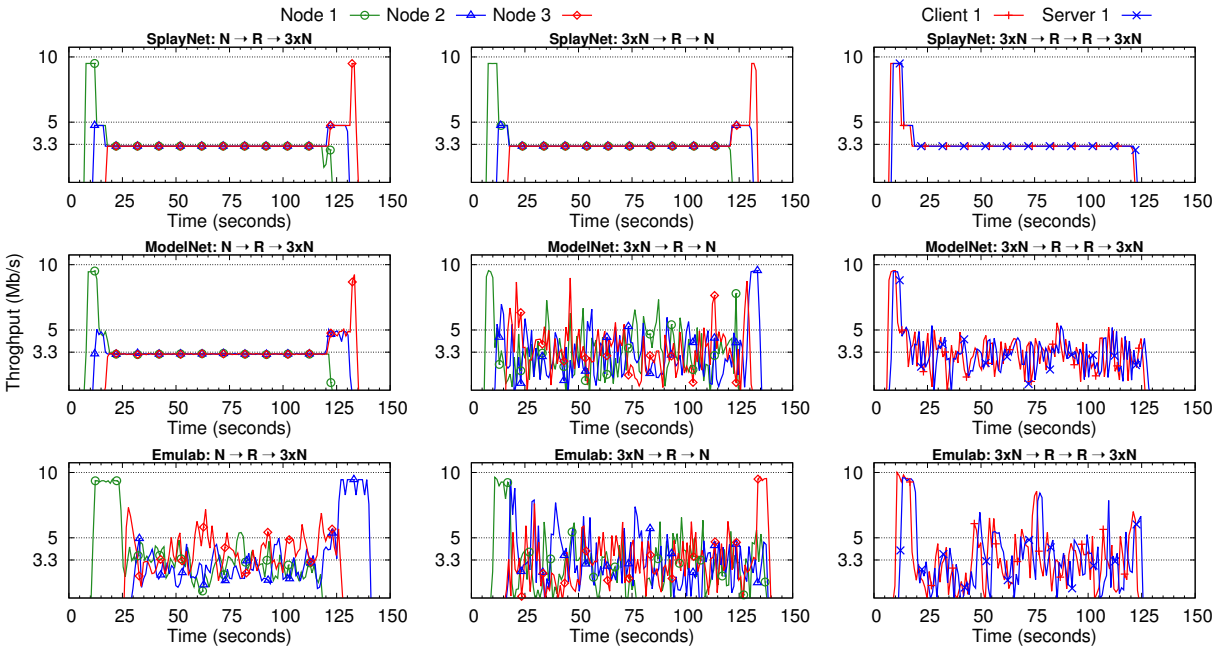
(c) Link bandwidth emulation.

Figure 3.6: Link latency and bandwidth emulation for a client-server RPC benchmark.

Bandwidth. Our second micro-benchmark evaluates the accuracy of the bandwidth emulation. We deploy the point-to-point topology of Figure 3.6.a (bottom) with two nodes connected by a single router. Link latencies are close to zero (bare latencies of the support cluster) to mitigate any bandwidth-delay-product effect [100, 101] and to allow the maximum theoretical throughput. Emulab and ModelNet's link queue sizes are configured with 100 slots. The client node continuously streams data to a server over a 10 Mb/s link via a pre-established TCP connection. Figure 3.6.c shows how the three systems let the application-level data stream, and emulated overhead, saturate the available link bandwidth up to the theoretical limits. ModelNet and Emulab present oscillations in the observed instantaneous throughput, while SplayNet provides a more steady download rate. This is a result of our choice of a decentralized, model-based network emulation that does not use kernel-level buffers at dedicated nodes. Oscillations are observed in real networks but to a much smaller extent than with ModelNet and Emulab. For the range of application of SplayNet (evaluation of



(a) Topologies for bandwidth emulation micro-benchmarks (N→R→3N, 3N→R→N, 3N→R→R→3N).



(b) Observed throughput at client nodes.

Figure 3.7: Bandwidth shaping accuracy.

networked protocols), the current model allows reproducibility between runs and between applications. We emphasize that oscillatory bandwidth allocation or reverse ACK traffic [102] can be integrated in the model without re-engineering the other elements of SplayNet.

We deploy more complex scenarios in order to evaluate the accuracy of SplayNet's bandwidth emulation when multiple clients concurrently stream data through common intermediate nodes. We use three topologies shown in Figure 3.7.a. Nodes are linked via 10 Mb/s links. Client nodes stream 50 MB of data to server nodes, competing for the bandwidth on the link that connects the client to the router (topology on the left, labeled N→R→3N), the link that connects the router to the server (topology on the center, labeled 3N→R→N), or the link between the two router nodes (topology on the right, labeled 3N→R→R→3N). Streams are started at intervals of 5 seconds. For the sake of clarity, in the case of 3N→R→R→3N, we only present the observed throughput at one client and one server. In order to isolate bandwidth emulation evaluation from the sharing model, we consider equal delays on all links (bare delay from the underlying network). The observed throughput in Figure 3.7.b indicates that SplayNet provides each stream with a fair amount of bandwidth even when competing with other streams, without dedicated machines to emulate routers and with no centralized traffic shaping orchestration. The results obtained with ModelNet and Emulab provide the applications an average throughput reasonably close to the expected value, but hardly reproducible from one run to another, or over the duration of an experiment.

Overheads. We evaluate our overhead emulation approach (Section 3.3.3) by measuring the amount of application-level bandwidth available to a simple application trying to saturate

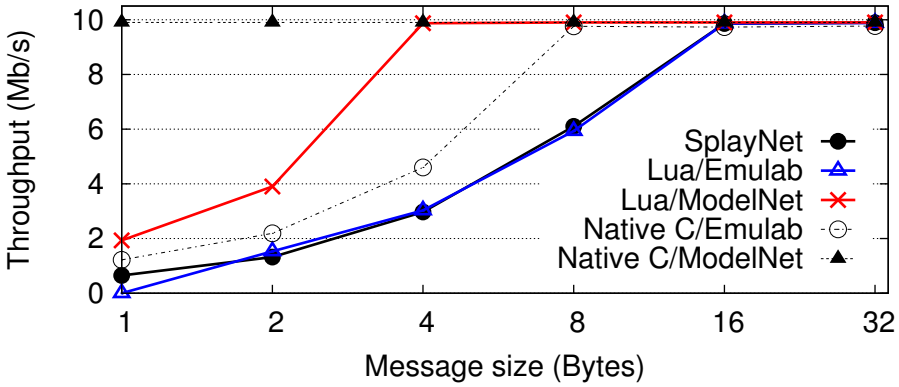


Figure 3.8: Impact of overheads on application-level bandwidth.

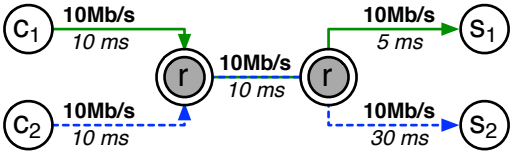
a N→R→R→N topology, by sending messages of increasing size. The middle link between both routers is the bottleneck with 10 Mb/s. We use TCP sockets with the TCP_NODELAY flag. This allows us to further stress the network. No buffering is used and messages are sent immediately. Each packet is subject to the overhead and small packets are not coalesced in larger ones (we prevent this from happening only for this experiment). The throughput is observed at the server side using the `iperf` tool. We use both a Lua-based application as well as a low-level C application.

Figure 3.8 presents the observed bandwidth for several settings. First, we observe that the use of the Lua language and LuaSocket library in SPLAY has some impact on the network performance. This is expected from the use of a high-level language and from the higher complexity in scheduling and context switching. Nonetheless, we observe that the network emulation itself yields similar overheads and accuracy between Emulab and SPLAYNET: the two lines overlap. The results with ModelNet, however, yield significantly higher throughput than expected for messages of size 1 to 16 bytes. The reason is that, to throttle the outgoing bandwidth rate, ModelNet issues ACKs with incorrect checksums back to the TCP sender, in order to mimic the non reception of traffic and trigger throttling. This produces a counter-reaction at the client, which jams packets that it needs to resend in a single ethernet frame. This effectively cancels the effects of the TCP_NODELAY flag and yields throughputs corresponding to sending large messages, preventing us from properly evaluating ModelNet overhead emulation accuracy.

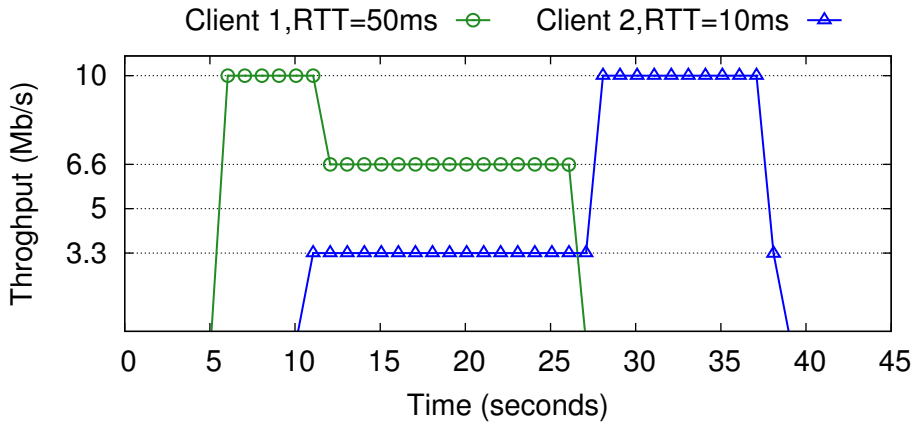
Link sharing. We now evaluate the effectiveness of the RTT-aware Max-Min link sharing model introduced in Section 3.3.3. We use the topology described by Figure 3.9.a and set up two flows from c_1 to s_1 and from c_2 to s_2. The r→r link is shared by the two flows and the maximal bandwidth achievable by both due to their bandwidth-delay product is greater than the link's capacity of 10 Mb/s. The first flow starts at second 5 while the second starts at second 10. As expected, when the intermediate link is traversed by both flows, its capacity is split according to the inverse of each flow's RTT: $\frac{50}{150} = \frac{2}{3}$ of 10 Mb/s for client 1 (∼6.66 Mb/s), and the remaining $\frac{1}{3}$ of 10 Mb/s for client 2 (∼3.33 Mb/s).

3.4.2 Macro-Benchmarks

For our set of macro-benchmarks, we deploy complete implementations of three representative distributed protocols, for which network emulation can be instrumental to evaluate their



(a) Topology



(b) Observed bandwidth at servers s_1 and s_2.

Figure 3.9: RTT-aware Max-Min sharing of 10 Mb/s bottleneck link.

performance. For the two first experiments, using the Chord [13] DHT and a collaborative multicast application [63], nodes are deployed on an emulated star topology where all end-nodes are connected through a single central inner router. All links from the end-nodes to the router are emulated at 10 Mb/s (symmetric) with 30 ms latency. The third experiment uses a *butterfly* topology and evaluates the impact of network coding [64] on the performance of an end-to-end data transmission.

Delay-sensitive: Chord DHT. Our first representative protocol is the Chord DHT [13]. After 20 nodes form a stabilized Chord ring, each node submits 50 queries for random keys. Figure 3.10 presents the CDF of the delays for all queries (left) and the CDF of the number of *hops* required by the queries to reach the node in charge of the key (right). The constructed rings do not perfectly overlap due to the nature of Chord node identifiers.[7] These results demonstrate similar behavior across all the testbeds in terms of latency emulation.

Bandwidth-sensitive: multicast. We now evaluate how SplayNet performs compared to ModelNet and Emulab for bandwidth-intensive protocols. We use a multicast protocol based on parallel n-ary trees [63]. We create n=4 distinct trees as done in SplitStream [103]. Each of the 20 nodes is an inner member in one tree and a leaf in the others. The data to transmit is split into 16 blocks of 2.5 MB each. Blocks are propagated in parallel along the 4 trees using a round-robin policy for tree selection. Figure 3.11 presents the CDF of the download completion time for all 4 trees at all nodes. The results obtained with the three systems indicate that the platforms offer comparable performance in terms of bandwidth emulation.

ModelNet, however, seems to provide less bandwidth than expected. To understand this behavior, we tried to fine-tune the ModelNet settings, in particular the emulator node's queue sizes. Figure 3.12 presents our results. SplayNet' bandwidth is in the range of

[7]Node identifiers are initialized by hashing their IP and port. Emulab does not allow choosing the network mask of the assigned machines.

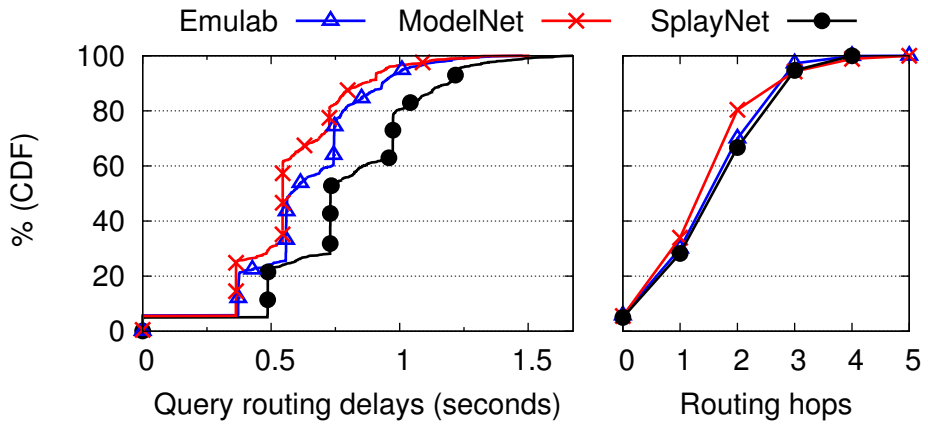


Figure 3.10: Routing in a 20 nodes Chord ring.

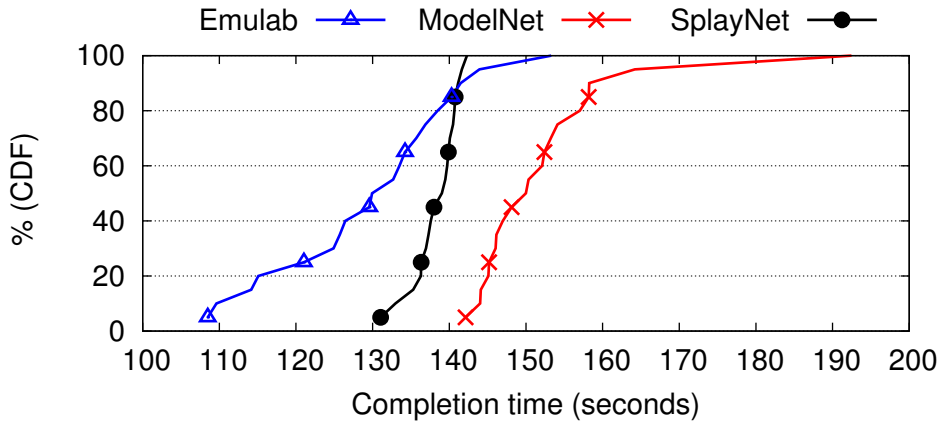


Figure 3.11: Multicast diffusion on n-ary trees.

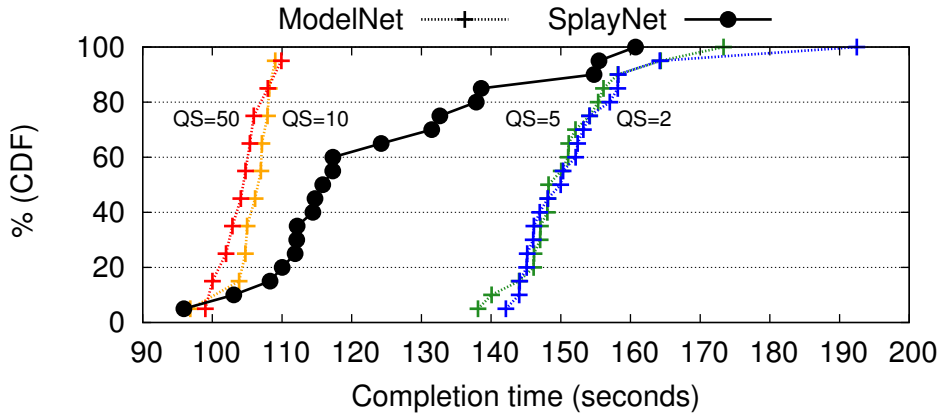
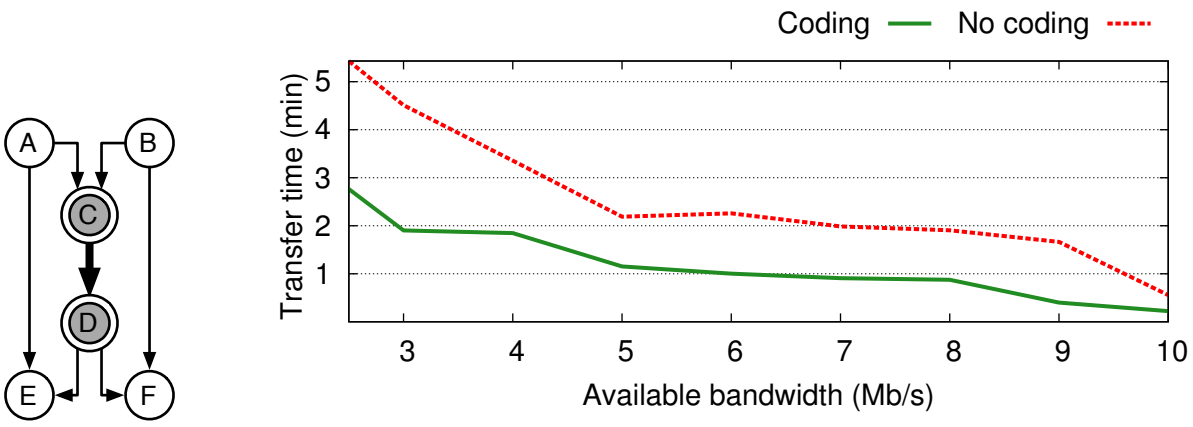


Figure 3.12: N-ary tree: impact of ModelNet's queue-size (QS) on completion time.

bandwidth obtained with ModelNet by tuning the queue sizes between 5 and 10, without any notable impact with larger or smaller queues. This indicates that ModelNet is particularly sensitive to the underlying system parameters, which may be difficult to set adequately.



(a) Topology

(b) Impact of C→D bandwidth on completion time

Figure 3.13: Network coding on a Butterfly [64].

Topology-sensitive: network coding. Our third macro-benchmark reproduces a classical problem for studying the effect of network coding, initially described in [64]. This experiment illustrates the benefits of emulating congestion between inner nodes of a topology. It also uses the ability of SplayNet to have application nodes attached to inner elements (routers) of the topology. As ModelNet and Emulab do not offer this possibility, we do not provide a comparison for this experiment. We consider the *butterfly* topology in Figure 3.13.a. Two servers, `A` and `B`, each stream 50 messages of 1 MB to two clients `E` and `F`. Every client must receive all messages from both sources. The two inner nodes, `C` and `D`, act as *application-level routers* and participate to the dissemination as message forwarders. All links but the central `C`→`D` link have a maximal bandwidth of 10 Mb/s. The bandwidth of the `C`→`D` link varies from 2.5 to 10 Mb/s and represents the bottleneck in the topology. The objective is to alleviate the impact of this bottleneck using network coding.

Without network coding, the transmission of messages from A and B compete for the bandwidth on the `C`→`D` link. Using network coding, when `C` receives messages m_a and m_b from `A` and `B`, it combines them—using an xor operation in our simple scenario—and sends the resulting message $m_a \oplus m_b$ to `D`, which in turns forwards it to both `E` and `F`. This allows reconstructing m_a from (m_b,$m_a \oplus m_b$), and m_b from (m_a,$m_a \oplus m_b$). The achievable bandwidth is increased as each message on the `C`→`D` link allows both `E` and `F` to obtain respectively a message from `B`'s and `A`'s stream. By tuning the bandwidth available on the `C`→`D` link, and testing with diverse (e.g., linear, random) coding strategies, researchers can explore the benefits of these techniques on a real deployment, as shown in Figure 3.13.b.

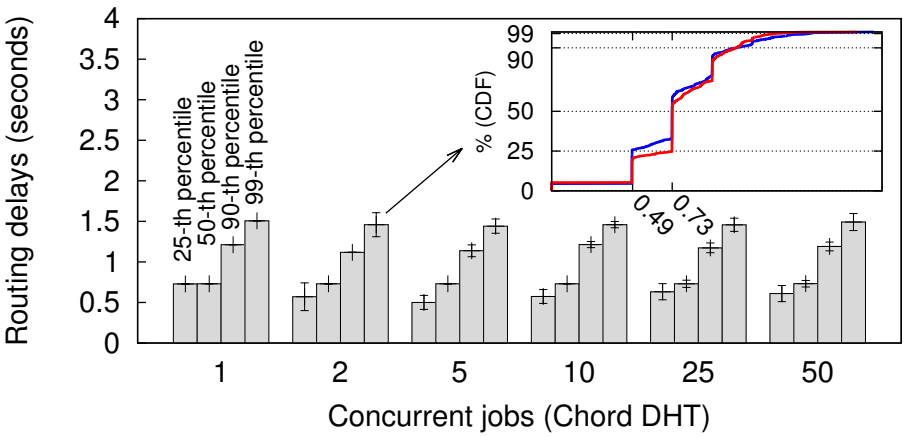


Figure 3.14: Impact of concurrent deployments on delay-sensitive protocol Chord.

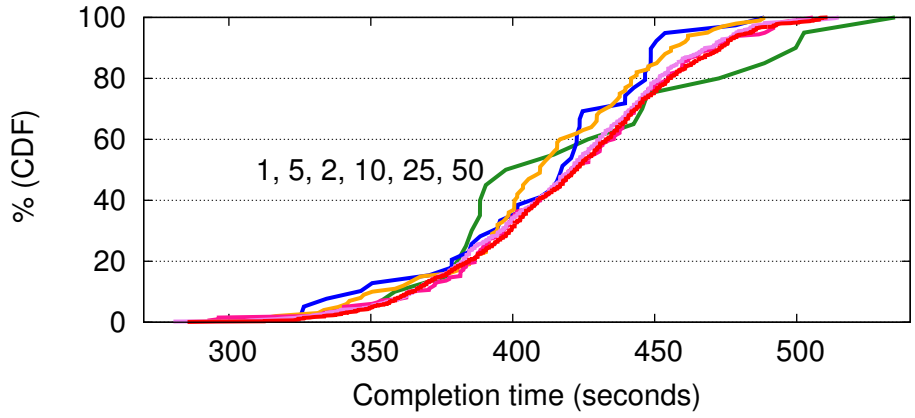
3.4.3 Concurrent Deployments

We now evaluate the impact of concurrent deployments in the same testbed on the emulation accuracy for both delay- and bandwidth-sensitive protocols. In these experiments, we use only 10 physical nodes of our cluster to enforce a high level of concurrency. Each individual deployment consists of 20 nodes in a star-like topology with 30 ms latency and 10 Mb/s bandwidth links. In the most extreme case of 50 concurrent jobs, up to 1,000 application nodes run simultaneously on the testbed. We start with a delay-sensitive application. Figure 3.14 presents the results of query routing delays when deploying up to 50 concurrent jobs, each running one instance of the Chord DHT. Each bar in a group of four presents a representative percentile (the first quartile, the median, the 90th and 99th percentile) of the routing delays for 50 random queries issued by each of the nodes. The inner graph shows the CDF of the routing delays for the queries issued by the nodes in the case of two concurrently deployed jobs, for which the percentiles give a compact representation. The standard deviation for each quantile is indicated on each bar. For instance, the median routing delay for two concurrent experiments are 0.49 s and 0.73 s, which yields an average median of 0.61 s and a standard deviation of 0.17 s. The small standard deviations and consistent quantiles confirm the lack of variation between the observed performances of concurrently deployed jobs.

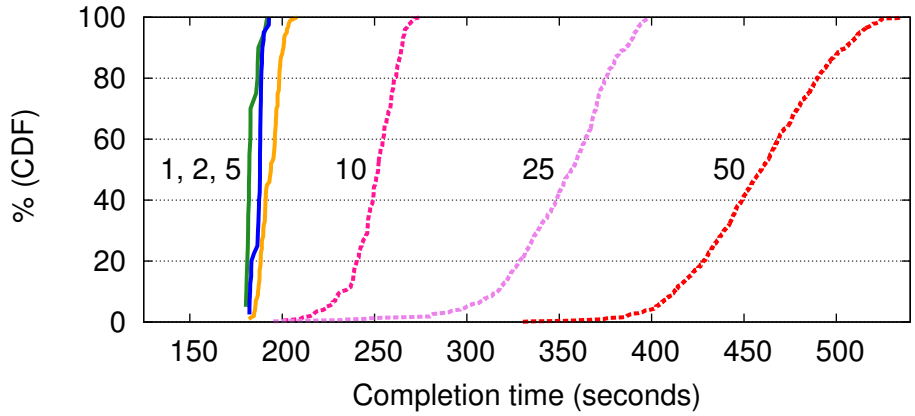
We continue by performing multiple concurrent deployments of a bandwidth-sensitive protocol, the parallel n-ary tree protocol previously described. Our objective is that concurrent experiments have little to no impact on one another, and in particular on the behavior of the protocol under test. The behavior of a set of protocols is represented by the CDF of the completion time for retrieving a file from the parallel trees. We use a star topology with low and high bandwidth requirements.

In low bandwidth settings, each link in the topology supports a bandwidth of 128 Kb/s and the transmitted file size is 2 MB. We observe in Figure 3.15.a that the deployment of 1 to 50 concurrent instances of the protocol have no impact on their performance, allowing to safely rely on a shared emulation testbed. The emulated traffic passing through each physical link of the cluster is below the threshold of 100 Mb/s we use in our experiments.

In high bandwidth settings, each link in the emulated topology has a requirement of 10 Mb/s and the transmitted file size is 40 MB. In this scenario, the maximal emulated traffic of 100 Mb/s per physical link allows selecting resources for at most 5 concurrent experiments in our 10 nodes cluster (100 application nodes in total). The selection algorithm actually



(a) 2 MB of data with 128 Kb/s links



(b) 40 MB of data with 10 Mb/s links

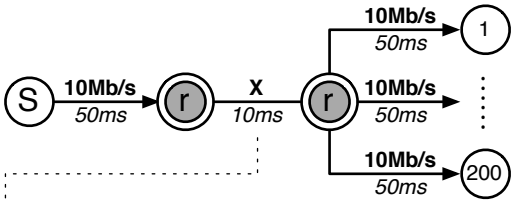
Figure 3.15: Concurrent n-ary tree deployments (the number of concurrent deployments is indicated next to the respective lines in the graphs).

prevents over-allocation of nodes. We illustrate its role by disabling it and running 1 to 50 concurrent experiments, as shown in Figure 3.15.b. Plain lines represent situations that are normally accepted by the selection mechanism, while dashed lines represent over-allocation that are prevented. The behavior of the protocols and topologies is consistent from 1 to 5 concurrently deployed topologies. With more, the distributions of reception times show higher values as concurrent deployments adversely impact one another, decreasing emulation accuracy for all of them. The threshold of 100 Mb/s per physical link is cluster-specific, but can be easily determined by automated probing mechanisms.

3.4.4 Scalability

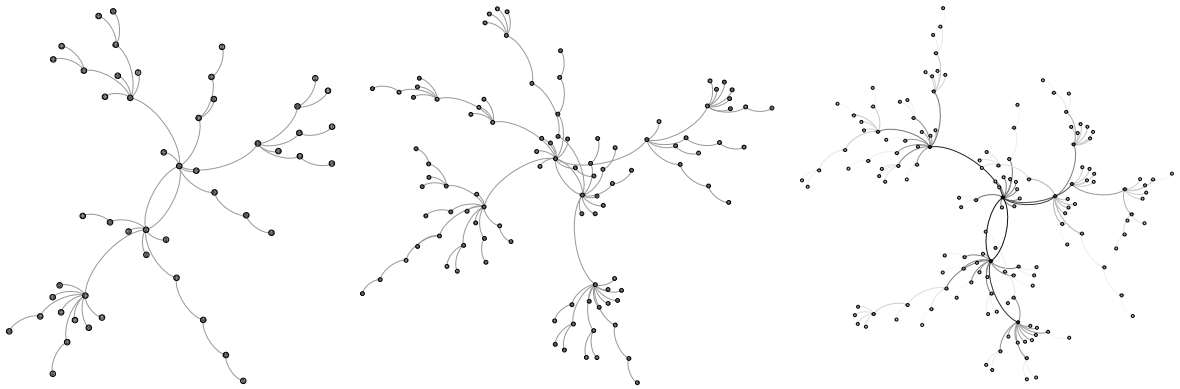
In this last set of experiments, we evaluate the accuracy and scalability of SplayNet when emulating large and complex topologies. We compare the accuracy of the emulation against "ideal" results obtained using a centralized and omniscient simulation. Based on the full list of exchanges, we can determine the exact congestion on inner links and decide on appropriate bandwidth allocation with no synchronization delay. The simulation uses the same mechanisms for deciding on congestions and bandwidth allocation (Section 3.3.3) but applies them to the full topology graph.

We first deploy a massive parallel upload scenario, illustrated by Figure 3.16 (top). A single source `S` uploads 1 MB of data to 200 clients through TCP connections. All links



bandwidth X	avg.	stdev.	ideal	accuracy
256 Kb/s	6742.77 s	17.99 s	6741.91 s	99.99%
512 Kb/s	3347.51 s	18.14 s	3370.95 s	99.3%
1024 Kb/s	1698.31 s	18.63 s	1685.47 s	99.24%

Figure 3.16: One-to-many parallel upload of 1 MB of data from 1 server node to 200 clients: average, standard deviation, ideal completion time and observed accuracy.



(a) 30 nodes, 20 routers. (b) 62 nodes, 38 routers. (c) 98 nodes, 52 routers.

Figure 3.17: Scale-free topologies construcuted using the preferential attachment method.

support 10 Mb/s except for the link denoted by `X`, which supports from 256 Kb/s to 1 Mb/s. All links from the second `r` to any end-node have the same characteristics. The bandwidth to each of the clients should thus be the same and the download times should be similar. This is indeed what we observe in the results in Figure 3.16 (bottom). The overall throughput is extremely close to the ideal, even though each end-node takes decisions about congestion on the `S`→`r` and `r`→`r` links independently.

Second, we use a set of three topologies of size 50, 100, and 150 nodes, constructed using the *preferential attachment* method [104]. We start with a single node and add new nodes one by one, each with one outgoing link. We pick the destination of that link such that the selection probability is proportional to each node's actual in-degree. This method yields *scale-free* networks, representative of the characteristics of Internet topologies, with distribution of the degrees following a power-law. Nodes with no incoming link act as application nodes while other nodes are routers. Due to the scale-free nature of the graph, a large majority of paths between end-nodes share common inner links in the topology. This is a challenge for the distributed congestion evaluation mechanism. Each link has a random delay in the [10:30] ms range. Bandwidth between routers is 1 Mb/s, and 10 Mb/s from end-nodes to their respective routers, to prevent the last link from being a bottleneck and to emphasize the effect of congestion on inner links. Figure 3.17 shows the topologies used for this experiment.

		flows/s		accuracy error (±%)			
nodes	**routers**	**avg.**	**time (s)**	**avg.**	**stdev.**	**min**	**max**
30	20	4.54	398.92	1.02	0.92	0.04	2.61
		9.78	719.11	3.89	2.14	0.08	8.15
62	38	7.49	400.85	3.45	2.12	0.65	8.52
		15.34	959.56	5.63	3.38	1.46	17.12
98	52	9.79	566.56	4.00	1.80	1.83	7.68
		19.09	1201.38	11.94	4.75	0.23	24.48

Table 3.2: Accuracy versus centralized simulation, on large scale-free topologies, of a randomized high-bandwidth communication workload.

We mimic a randomized bandwidth-sensitive communication workload. Some application nodes initiate a single communication of 10 MB of data over TCP to a randomly selected other node. This is similar to what would happen for instance in a BitTorrent dissemination. For each topology, we use two workloads: a light and a heavy one (first and second line of Table 3.2, respectively), which differ in particular in the number of (concurrent) exchanges. The last four columns present the statistics for the accuracy, that is, the variation over the ideal simulation for the same exchanges. The average accuracy ranges from ±1.02% to ±11.94%, with only small variations across all flows and in all cases, i.e., a low standard deviation. Minimal and maximal inaccuracy is particularly low for the smallest graph and remains reasonable for the two others, well in the usability range for large-scale network emulation. We were not able to deploy the same experiment on Emulab due to the low number of available nodes on this platform.

3.5 Summary

Network emulation allows researchers to evaluate distributed applications by deploying them in a variety of network conditions and topologies. Previous solutions often relied on dedicated machines to shape the network traffic across the nodes involved in an experiment, and did not allow the concurrent deployment of different network topologies on the same nodes of a testbed.

This chapter introduced SplayNet, an integrated user-space network emulation framework. SplayNet uses a distributed orchestration protocol to emulate congestion at inner nodes in a decentralized manner and without instantiating these inner nodes on physical machines. It allows the deployment of multiple experiments, each under different network emulation conditions, and running concurrently on the same set of machines. SplayNet offers equivalent performance to state-of-the-art systems, both in terms of latency emulation and bandwidth shaping accuracy. It has shown to scale well for concurrent deployments of real-world distributed protocols and large topologies.

Chapter 4

Brisa: Lightweight, Efficient, Robust Epidemic Dissemination

The previous two chapters presented Whisper (Chapter 2) and SplayNet (Chapter 3). These two systems presented solutions to overcome network topology restrictions (the former) or to emulate network topology conditions (the latter). This chapter and the following one tackle the problem of building topology-aware distributed systems. In particular, we introduce a data dissemination system for large-scale systems that support highly dynamic deployment contexts, such as large-scale networks, by relying on lightweight gossip-based protocols. The topology-awareness aspect of this system is materialized by its ability to organize the virtual-topology, application-level routing layer according to different configuration criteria.

4.1 Introduction

We live in a digital era whose foundations rely on the production, dissemination, and consumption of data. The rate at which content is produced is constantly increasing [105], putting pressure on dissemination systems able to deliver the data to its intended consumers. Examples include the distribution of digital media (e.g., music, news feeds) on the Internet [106] or software updates in a datacenter infrastructure [107].

On account of its importance, significant research has been dedicated to conceiving efficient and robust data dissemination systems [17, 103, 108–110]. Unfortunately, both design vectors, efficiency and robustness, are often addressed disjointly: either by a highly efficient structure based on trees like in [111] or by a highly robust unstructured epidemic or gossip-based approach such as [110].

However, under churn and faults, the rigid structure that makes the tree efficient must be rebuilt constantly, hindering robust dissemination and continuity of service, and significantly increasing delays for all nodes that lie in the subtree rooted at a failed node. These reconstruction delays moreover accumulate along the path to leaves, when multiple faults occur during a dissemination. Furthermore, only interior nodes contribute to the dissemination effort while resources of leaf nodes remain unused which leads to poor load balancing.

On the other hand, epidemic-based dissemination systems rely on redundancy instead of structure to offer guarantees on the delivery of data to all participants [17, 110]. Gossip-based dissemination was initially proposed in the context of database replica synchronization in the ClearingHouse project [112]. The transmission of several copies of the same message to random nodes enables epidemic-based systems to be oblivious to faults and churn, as the

same message will be received through different paths. The cost is increased bandwidth and processor usage due to the transmission and processing of duplicates. Epidemic principles have also been used elsewhere to build robust and scalable distributed systems components such as membership [30, 113, 114] (see also Section 1.3.2) and failure detection [28] services, or indexing mechanisms [27, 115]. As long as (1) the graph induced by the (partial) views offered by the membership service is connected and (2) all nodes have at least one incoming link, dissemination can trivially be achieved by flooding.

Several proposals try to overcome the weakness of each approach by combining them. For instance, SplitStream [103] builds several trees and stripes the application data among them to distribute the load and increase robustness to faults. The management overhead of such approaches is however non-negligible under churn due to its structured nature. Others like TAG [116] and MON [109] build an overlay and a tree and pull application data through them. Pulling data is an effective mechanism to avoid duplicates which unfortunately comes at the cost of increased latency and requires receivers to periodically poll senders. Our approach also uses overlays and trees but in such a way that the maintenance cost of the trees, even under churn, is comparable to that of simple overlay. Moreover, due to the way trees are built, data is pushed through them thus minimizing latency.

Contributions

In this chapter we present Brisa, an efficient, robust and scalable data dissemination system. Brisa leverages the robustness and scalability of an epidemic substrate to build efficient dissemination structures that are correct, i.e., cover all nodes, by construction. Such structures are built in a distributed fashion with local knowledge only and with minimal overhead. Brisa has been designed in a way that upon failures or churn, trees are easily and rapidly repaired thanks to the underlying epidemic substrate that acts as a safety net. As dissemination structures we consider trees, directed acyclic graphs (DAGs) and forests of trees. We evaluated Brisa on PlanetLab [62] and on a local cluster comparing it with state-of-the-art data dissemination systems from the literature.

Outline

The remaining of this chapter is organized as follows. Section 4.2 describes the design of Brisa and Section 4.3 presents the experimental evaluation. In Section 4.4 we discuss related work and finally Section 4.5 concludes the chapter.

4.2 Brisa

In this section, we describe the design of Brisa. Brisa relies on an underlying peer sampling service (PSS). The main concepts of a PSS were previously introduced in Chapter 1.3.2. In the following Section 4.2.1 we discuss the requirements and the guarantees it provides in the context of Brisa. Then, we introduce in Section 4.2.2 the key design principles of the Brisa protocol and how the dissemination structures are constructed. Finally, we show how Brisa deals with dynamism, generalize the construction of dissemination structures with desirable efficiency/robustness criteria and discuss the creation of multiple dissemination structures.

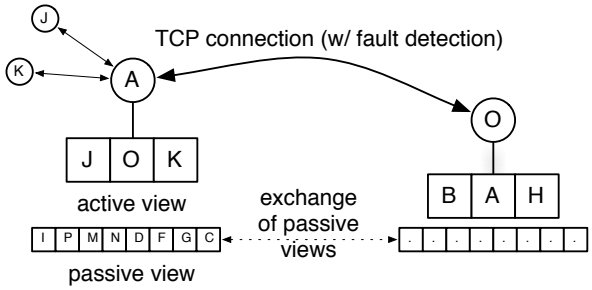


Figure 4.1: HyParView [114]: view maintenance.

4.2.1 Peer Sampling Service Layer

We assume an underlying PSS [30] that provides each node with a *view*, i.e., a set of non-faulty nodes chosen at random from the entire network. The basic principles and mechanisms of a PSS are described in Section 1.3.2. Here we provide more details in particular with respect to the update mechanisms for the views mantained at each node. The update of the views can be either continuous (*proactive* peer sampling) or happen only when a node fails or a new one joins the system (*reactive* peer sampling). In the *proactive* case, nodes periodically share their views with their neighbors regardless of the actual need to replace failed entries, resulting in each view being a continuous stream of node samples from the network. Examples of proactive PSSs include Cyclon [31] and Newscast [117]. In the *reactive* case, the view is kept unchanged unless some of its entries need to be updated, i.e., for replacing a failed node or for accommodating a node joining the system. Typical examples include Scamp [113], Araneola [118] and HyParView [114].

We rely on a reactive PSS and more specifically on HyParView [114]. The motivation for this choice comes from the additional stability of reactive approaches, which simplifies the process of creating efficient and correct dissemination structures. In short, HyParView maintains two views at each node: a larger *passive* view and a smaller *active* view (see Figure 4.1). Only the active view containing the node's neighbors is exposed to the application and in particular to Brisa. The passive view is maintained in a proactive manner by periodic exchanges and shuffling of passive views with randomly selected neighbors, that are also selected from the passive view itself. The entries in the active view are managed in a reactive manner: a neighbor in this view only changes upon failures, or for accommodating a newly joined node. An opened TCP connection is maintained with each of the nodes in the active view for communication efficiency, in particular, latency. Due to the limited size of the active view, efficient heartbeat-based fault detection can be used for all of its members. Upon detection of a failed neighbor, a replacement node is selected from the passive view and moved to the active view. When the active view is full and a new node attempts to join, a random node is removed from the active view to accommodate the joiner.

An important aspect of HyParView is that links with neighbors are *bidirectional*. If node A has node B in its active view, then B also has A as its neighbor. In a connected overlay, using bidirectional links allows us to ensure that messages disseminated by flooding will reach all the nodes in the system without requiring anti-entropy mechanisms where nodes periodically poll other nodes for the content they might have missed [112]. A node receiving a message *for the first time* from a neighbor simply propagates it to all its other neighbors.

In order to avoid chain reactions due to the massive number of joins when bootstrapping the system (node A's view size is full so it removes node B, B also removes A from its view

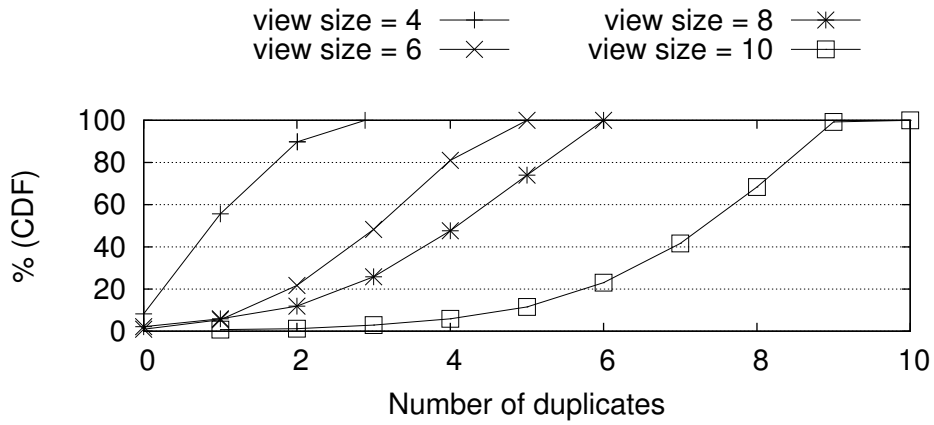


Figure 4.2: Distribution of duplicates per message for each node for 500 messages in a 512 nodes HyParView network for various active view sizes.

and promotes a node C from its active view, C must add B to its view and thus remove an existing one as its active view is already full, removing D and so on and so forth), we allow the active view size to grow past the configured value by a given *expansion factor*. Node evictions do not result in replacements when the view size is between the target view size and this size times the expansion factor. We used an *expansion factor* of 2 throughout the evaluation. The impact on the actual view sizes is limited as shown later in the analysis of the degree distribution (Section 4.3.1, Figure 4.7).

Flooding is ensured to reach all nodes as long as no node in the system has an active view containing only failed nodes. The larger the active views the smaller the chances for this to occur. However, the larger the view, the larger the number of relayed messages and consequently the number of duplicate receptions. As a concrete example, Figure 4.2 presents the cumulative distribution function (CDF) of the number of duplicates during the dissemination of 500 messages over a 512 nodes HyParView network for different view sizes. We observe that as the size of the view grows, nodes quickly receive large amounts of duplicate messages. For instance, half of the nodes receive more than one duplicate with a view size of 4, while they receive more than 7 duplicates with a view size of 10.

Brisa develops on top of HyParView. It takes advantage of the connectivity guarantee that can tolerate up to 80% node failures [114] to emerge efficient dissemination structures that eliminate (or considerably reduce) the number of duplicates, while keeping the robustness offered by the underlying PSS.

4.2.2 Rationale

The objective of Brisa is to support the efficient, robust and scalable dissemination of a stream of messages from one or several sources to the entire network. Efficiency relates primarily to the limitation of duplicate message transmissions that waste bandwidth and processor resources. On top of that, Brisa can consider additional efficiency criteria, namely: the reduction of the end-to-end delay (dissemination time from the source to the last receiver) and network efficiency (ratio between the delay for receiving a message through Brisa as compared to a hypothetical direct communication from the source). Robustness relates to fault tolerance: dissemination should progress despite the inactivity of some nodes (failure or disconnection) and the system should be able to rapidly detect and mask such faults. Finally,

Brisa scales to very large networks, because the view size is kept small and under strict control by the PSS thus preventing the load at any node to grow linearly with the system size.

The main idea behind Brisa stems from the observation that it is the *possibility* of receiving messages through multiple paths that makes epidemic-based approaches robust, not necessarily the actual data transmission. Our goal is to therefore to limit or even eliminate duplicate transmissions while maintaining the *possibility* of receiving the messages through multiple paths. Such possibility is given by the view provided by the PSS which contains a set of potential senders. From this set, Brisa selects one or more to perform the actual data transmission thus materializing the possibility into a concrete delivery.

Based on this selection, Brisa automatically emerges dissemination structures on top of the undirected HyParView overlay. Such structures are oriented and can be either trees, by restricting the inbound neighbors of every node to a single node (parent), or directed acyclic graphs (DAG) by allowing multiple parents for each node. The creation of a structure is performed by local and unilateral decisions made by the nodes about the set of neighbors that should be *active* and actually relay inbound traffic and those that should be *inactive*. In the case of a tree the reception of duplicates is effectively eliminated; in a DAG, it is selectively reduced.

The resulting dissemination structure must ensure complete disseminations, i.e. that all nodes receive all messages. To that end, we must ensure that it does not contain a non-connected sub-graph that would not receive the message from the other components of the structure. This property is ensured by enforcing the absence of *cycles*. In fact avoiding cycles is the main concern when determining the set of active and inactive neighbors of a node. In the following sections, we first describe how the emergence of a single tree is achieved in Brisa, then generalize the approach to DAGs, and finally delineate the use of forest of trees.

4.2.3 Emergence of a Dissemination Structure

The emergence of Brisa's dissemination structures is part of the natural operation of the system and is based on the reception of duplicates. Nodes start with all the links active and thus the initial dissemination structure exactly matches the HyParView overlay. These links form a graph that serves as the basis for the construction of a Brisa dissemination structure. Initially, a source node sends the first message of the stream to all its neighbors. Nodes receiving the message for the first time simply forward it to all the nodes in their view because all links are active, effectively flooding the network.

This flooding operation reaches all nodes, given the connected and bidirectional nature of the overlay provided by HyParView. During the initial flood, nodes receive the message from a number of different neighbors. Out of these sources, each node autonomously selects one as its parent in the dissemination structure and sends a deactivation message to all the others. Future messages in the stream will then be received only from the selected parent node. The selection is achieved by the use of a *link deactivation* mechanism and follows one of the selection strategies presented in Section 4.2.5. To emerge a tree each node simply needs to prune out *all but one* of its inbound links. Note that the bootstrap can also be done by injecting an empty message (without payload) in the system if the initial flood of an application message poses bandwidth concerns.

It is important to note that deactivating a link does not imply removing the corresponding entry from the HyParView active view. The overlay constructed by the PSS remains available and is used both as a provision of nodes for reparations upon failure, or as

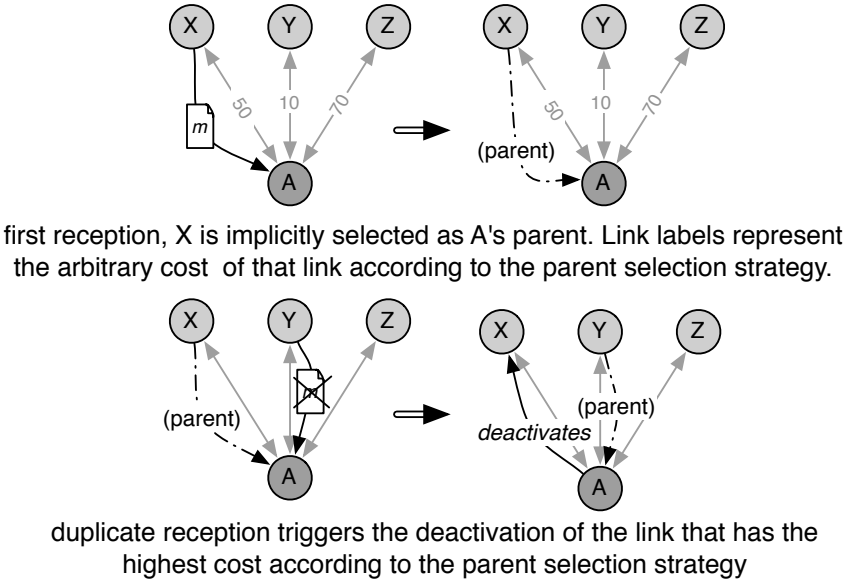


Figure 4.3: Reception of a duplicate and deactivation of one link, for a tree BRISA structure. Depending on the parent selection strategy, the deactivated link can be the previous parent or the node sending the duplicate.

a fallback for dissemination when reparation is temporarily not possible. Figure 4.3 presents the principle of the link deactivation mechanism for constructing a tree. Initially, links from nodes X, Y, and Z belonging to node A's view and are all *active*. The first reception of a message from node X results in node A considering X as its parent. A subsequent reception of a duplicate from node Y or Z triggers the link deactivation mechanism. As only one inbound link should be active, node A needs to deactivate either the link from node X or the link from node Y. In our example, as the cost of Y is lower X selects it as its parent and deactivates the previously active link from X.

There are three guiding principles for deciding which link to deactivate. First, the dissemination structure must not contain cycles. Second, it must seek to meet the target number of parents for each node (one for the tree structures, more when generalizing to DAGs). Finally, when both conditions are met, the parent selection strategy chooses the new parent based on different criteria for shaping the dissemination structure (Section 4.2.5).

4.2.4 Preventing Cycles

A mandatory condition for selecting a parent node is that it does not yield a cycle in the dissemination structure. This means that the potential parent of a node N does not receive the stream directly or indirectly from N itself. For a tree this implies that the parent of N must not appear in the sub-tree rooted at N.

To verify this condition each node piggybacks on the application messages the node identifiers in the path from the source to itself. When selecting its parent, a node N rejects those candidates whose message path to the source includes N itself. This is illustrated in Figure 4.4, where grey nodes are not eligible as parents of node N. It is important to note that the overhead of path embedding is minimal and very attractive when compared to probabilistic inclusion structure such as Bloom filters [119]. As a matter of fact, the size of the embedded path is bounded by the tree height, which is expected to be $O(log_b(N))$ where

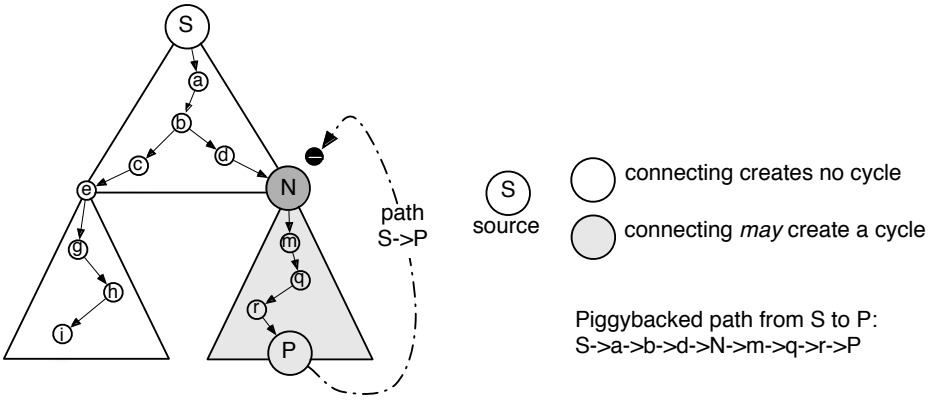


Figure 4.4: Avoiding creating a cycle for a tree, by checking that the node N is not in the dissemination path to the potential parent.

N is the system size and b the active view size. For instance, in a system with 1×10^6 nodes with an active view size of 8, the average tree height is $log_8(1 \times 10^6) \approx 7$. This bounds the maximum metadata size a message needs to carry which, assuming a 48 bit (ip,port) pair as unique identifier, is only 336 $(7 * 48)$ bits. A bloom filter, to ensure a reasonable false positive probability to avoid detecting cycles where there is none, would require about the same number, or more, bits. Taking into account the metadata size required, the fact that path embedding is exact (false positive probability is zero) and the computational overhead associated with Bloom filters (which requires computing several hashes), path embedding presents many advantages over Bloom filters.

The detection of cycles is not only done during the initial flooding phase: a node that detects a cycle from a parent simply makes the link from that parent inactive and selects a new parent using the regular selection mechanism or the fallback to flooding as we describe later in Section 4.2.6.

4.2.5 Parent Selection Strategies

From N's eligible parents (that is, those not having N in the path followed by the messages from the source), Brisa selects one according to the following strategies:

1. First-come first-picked. The node sending the first received message is selected as parent, all subsequent duplicates received trigger the deactivation of the incoming link.

2. Delay-aware. This strategy considers the round-trip time between N and the candidate nodes. The one with the lowest delay is selected as parent. We leverage the periodic keep-alive messages that are exchanged by the nodes in the active views at the HyParView level to measure round-trip times.

A simple optimization is available when building a dissemination tree using the first-come first-picked strategy: the deactivation of links can be symmetric. Supposing node A receives a message first from node B and then from node C, A will pick the link from B and send a deactivate message to C. But it can further mark its outgoing link to C as inactive as A knows it will not be not eligible as parent for C, as C already received the message first.

4.2.6 Dynamism

The insertion and removal of nodes in the system is handled by the underlying PSS. A new node joins by contacting a node already in the system. The new node is provided with an active view of the same size as the node's contact point. It is then inserted in the active views of the associated nodes. Brisa automatically marks links to new nodes as active. As a result, the joining node will have all its inbound links marked as active and will receive its first message multiple times. All that remains is to select its parent(s) according to the mechanism discussed previously.

The detection of node failures is performed at the level of the active view, by exchanging periodic keep-alive messages over the established TCP connections, or when a node fails to acknowledge the reception of a transmission (as detected by the TCP flow control for that link). When a node notices that one of its neighbors is removed from the active view (due to a failure), it first checks if that neighbor was a parent. If that is not the case, the removal can be ignored. Otherwise, the node needs to find a replacement parent using one of two strategies. It first attempts a *soft repair* by trying to select as parent one of the remaining neighbors. A simple approach is to reactivate all its inbound links and proceed with the normal parent selection process. This can however be optimized by leveraging the keep-alive messages used for monitoring the active view at the PSS level and piggyback up-to-date information required by the parent selection procedure. If a suitable parent is found then its inbound link is directly re-activated. Note that this mechanism uses local knowledge only and requires a single message exchange being thus very fast and efficient. Furthermore, as shown later in the evaluation, almost all repairs can be done using the soft repair strategy resulting in minimal disruptions and very fast recovery of the dissemination structure (Section 4.3.3).

If no replacement parent exists in the active view, we resort to a *hard repair* that uses the underlying flooding approach for rebuilding part of the dissemination structure. The orphan node first re-activates all its incoming links and considers itself a fresh node by forgetting its position in the cycle detection mechanism. This allows the orphan node to take any of its neighbors as a parent. To ensure the tree remains connected, it is necessary to rebuild the incoming links for a part of the structure rooted at that orphan node. The need to repair a portion of the tree is detected by the children of the orphan node when they receive an activation request from their (former) parent. Those nodes proceed then with the local repair attempting first a soft repair and if not possible resorting to a hard repair. We note that the effects of the hard repair are limited to a small portion of the tree and in practice stop as soon as a node can find a suitable parent in its active view. Besides, the former parent will receive subsequent messages from the children (remember the parent activated that link) and may effectively exchange roles. The number of nodes affected by a hard repair is independent of the position of the original orphan node in the tree: it only depends on nodes in the sub-tree finding a suitable replacement parent, which is independent of the position of the original orphaned node.

Finally, nodes can compensate message loss during recovery by directly asking its newly found parent to send the missing ones. Since parent recovery is quick (Section 4.3.3) the number of messages each parent needs to buffer is small. Nonetheless more complex approaches such as [120] could still be used to ensure nodes buffer messages for long enough to allow recovery.

We note that the only requirement for trees to be repaired is the existence of some neighbors at the PSS level which implies that the graph induced by the PSS views is connected. One of those neighbors is then chosen by Brisa's repair mechanism as the new parent. It is

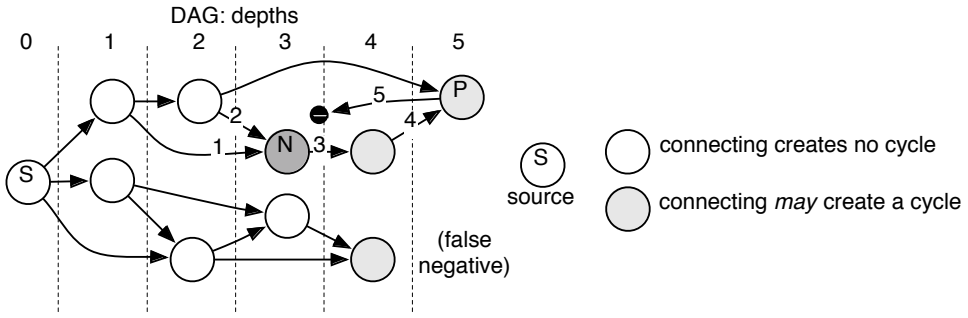


Figure 4.5: Avoiding creating a cycle for a DAG, by checking that the level of the potential parent is less than or equal to the level of the node.

important to note that PSSs in general, and HyParView in particular, are very robust to disconnections and able to maintain connectivity even under massive failures [114]. It follows thus that Brisa is also able to overcome the failure of a great portion of the network and eventually reestablish tree connectivity.

4.2.7 Generalized Dissemination Structures

To enhance service continuity under failures and churn, Brisa can generalize the tree structure to directed acyclic graphics (DAGs) by having each node being served by several parents instead of only one. In this way, a node that sees one of its parents fail can seamlessly keep receiving the flow of messages without the need to first undergo through the parent recovery process. This is attained at the cost of handling a controlled level of duplicate messages.

The establishment of a DAG basically involves making a number $p > 1$ of inbound links active in such a way that cycles are avoided. The technique to prevent cycles we used for trees is however unfeasible in the case of DAGs due to the amount of control information required to be exchanged. Indeed, a node in the n^{th} level of the tree requires a set of n node identifiers to define the path from the stream source to itself, while for a DAG with p parents per node this set at level n could reach $p^{n+1} - 1$ should all paths be non-overlapping.

Conversely, for DAGs, we use an approximate quantitative approach that does not include the nodes identifiers but just the *depth* each node is in the DAG as illustrated by Figure 4.5.

The source node is at depth 0 and every message carries its sender's depth encoded by a single integer. Initially, the depth of a node N is undefined and, upon reception of its first message from a node with depth $i - 1$, N places itself at depth i. From then on, N can select parents, and thus receive messages, from nodes at any depth not greater than i. Should N receive a message from a node at depth i (its current depth) then N moves to depth $i + 1$ and immediately updates its downstream children nodes accordingly. Similar to the technique we used for trees, it is clear that any node M served directly or transitively by node N will be at a depth strictly greater than N. Therefore, M cannot become a parent of N and yield a cycle.

As mentioned, the technique is however approximate because it can yield false negatives by discarding valid potential parents, as illustrated in Figure 4.5. Any two paths (rooted at S) are likely to be labeled similarly with respect to depths. Since the tagging is purely quantitative, a node from one path may be dismissed as a potential parent of a node in another path despite the paths being causally unrelated. An alternative is to rely on Bloom filters to maintain the set of nodes that need to be excluded for the parent selection process.

However, as for trees this a costly technique when compared to the simplicity and efficiency of depth encoding. In our experiments, nodes are able to obtain the desired number of parents, thus we consider this approach an attractive alternative when compared to the cost of both an exact predictor (path embedding) and of a probabilistic one (Bloom filters).

After determining the set of potential parents with the above strategy all that remains is selecting the *best* ones by using the parent selection strategies presented in Section 4.2.5.

4.2.8 Multiple Dissemination Structures

So far we discussed the creation of a single dissemination structure, be it a tree or a DAG. In the remainder of this section we motivate and describe the support for multiple dissemination structures. For clarity of explanation, we focus on trees but the same principles apply to DAGs.

There are several cases where it is interesting to support more than one tree, for instance if the source needs to split the content across several trees as in SplitStream [103] or to apply network coding techniques, or simply if there are several sources in the system. Moreover, the use of multiple trees enables a better use of system resources as more nodes can contribute to the dissemination effort. This is because when using a single tree, the leaf nodes, which are a big portion of the system, do not upload data and thus their capacity is not used. Supporting several sources can be done by building a single tree rooted at a *rendezvous* node that acts on behalf of all sources as in Scribe [108]. This design suffers however from a bottleneck in the *rendezvous* node and fails to take advantage of the upload bandwidth available at leaf nodes.

Therefore, we consider instead the creation of several independent trees. In Brisa, a tree is simply given by the set of active and inactive links that each node locally maintains. As a direct consequence, a node willing to maintain several independent trees simply needs to manage distinct sets of such links, one set for each of the trees in the system. Each tree is uniquely identified by a *flowId* generated by the tree source at construction time. Note that as, by assumption, each node has a unique id, it is straightforward to generate unique *flowIds*, for instance by concatenating the node id with a local sequence number. The source then tags all application messages with its *flowId*, enabling other nodes to uniquely assign the messages to the appropriate tree. Upon reception of a message from an unknown *flowId*, a node locally creates a new set of active and inactive links dedicated to managing that tree and proceeds as detailed in Section 4.2.3.

This approach is very lightweight as it requires the maintenance of a small local state, yet due to the inherent randomness in tree creation enables a much more efficient use of the overall upload bandwidth as few nodes are leaf in all trees (as we show in Section 4.3.4, Figure 4.13).

Nonetheless, from a design point of view, we observe that the state each node needs to maintain grows linearly with the number of trees in the system. To mitigate this, we designed a tree reusing strategy that can be used when the number of trees grows. The base idea is very simple: instead of creating a new tree, a node simply reuses one it already knows to disseminate its messages. To this end, the node analyses the trees it knows and if it is close enough to the root of any tree according to *reuseDepth*, a protocol parameter, it uses that tree's *flowId* instead of creating a new one. Note that, due to path embedding, a nodes always knows its position in all trees it belongs to, so computing the distance to any root is inexpensive and requires only local knowledge. By reusing an existing *flowId*, the messages created by that node will simply be relayed through the existing tree with no

further overhead. However, as the source node is not located at the root of the tree anymore, it is necessary to relay message upward in the tree, to ensure completeness. This is easily achieved by adding an *upward* flag to the message, implying that nodes need to relay those messages not only to their children but also to their parents. Another option would be to directly send the message to the root of the tree which would act as a *rendezvous* node. We note that the latter shows less bottlenecks problems than Scribe as it considers several *rendezvous* nodes, one for each existing tree, instead of just one. While simple, this strategy is very effective at reducing the number of total trees and the associated overhead. Obviously, reusing trees can have contradictory goals with the creation of multiple disjoint trees, e.g., as in SplitStream or as shown in our evaluation Section 2.5. In these cases, the goal is to create multiple disjoint trees from a single source, in order for leaves in a tree to act as interior nodes in the other, and reversely, in order to balance the load of the dissemination of a stream that is split among the trees. Tree reusing can limit the benefit of this approach, leaves remaining leaves in multiple trees and keeping the dissemination load unbalanced. Nonetheless, tree reusing can still be beneficial, between trees that are used for different streams. Tree reusing shall only be prevented for the trees of a given stream. In this case, each such tree is marked with the identity of the other trees from the stream, and reusing is disabled for those trees in the reusing decision process.

4.3 Evaluation

In this section we evaluate Brisa on two different testbeds: (1) a local cluster of 15 computers equipped each with 2.2 GHz Core 2 Duo CPU and 2 GB of RAM and connected by a 1 Gbps switched network, supporting up to 512 Brisa nodes and (2) a slice of up to 200 nodes on the global-scale PlanetLab [62] testbed. The prototype uses Splay (Section 1.3.1).

The evaluation is focused on the aspects that drove Brisa's design: efficiency and robustness. For each experiment and unless otherwise stated, we bootstrap the system with the specified number of nodes using the first-come first-picked strategy with an expansion factor of two, randomly choose a node to be the source across all the experiment and then have it inject 500 messages at a rate of 5 per second, taking measurements as appropriate. The message payload is an opaque random bit string with the specified size.

We start with a preliminary study, in Section 4.3.1, on the structural properties of the dissemination structures created by Brisa as those properties impose well-known bounds in resource usage and dissemination time. Then, in Section 4.3.2 we inspect the network properties of Brisa, namely bandwidth consumption and routing delays and analyze the results according to the structural properties. Next we evaluate the behavior of Brisa under churn in Section 4.3.3, and with multiple trees in Section 4.3.4. Finally, in Section 4.3.5, we compare Brisa with other approaches.

4.3.1 Structural properties

We first study the shape of the structures generated by Brisa, namely trees and DAGs with 2 parents. The shape (depth and degree), imposes constraints on latency and on the distribution of the dissemination effort. Results for each configuration are obtained after building the respective structure and letting it stabilize completely. The reason for using this basic strategy is twofold: i) a naive strategy helps to better understand the basic behavior of Brisa thus serving as a baseline for more elaborate strategies and ii) the limited number of physical nodes hides significant differences on the observation of structural properties changes

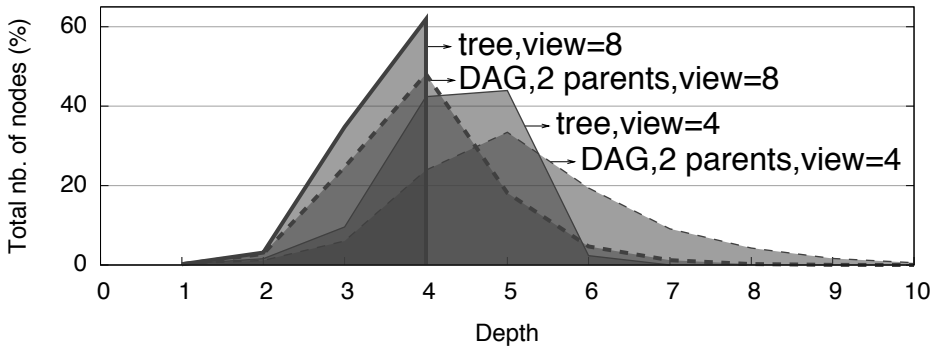


Figure 4.6: Depth distribution for 512 node (first-come first-picked strategy).

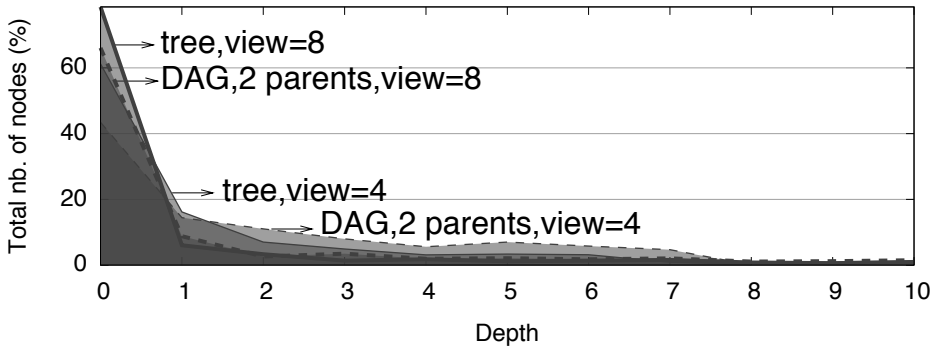


Figure 4.7: Degree distribution for 512 node (first-come first-picked strategy).

that are better observed at larger scales. Depth places a lower bound on the dissemination time due to the cost of traversing several intermediate nodes and thus should be kept as low as possible. Figure 4.6 presents the depth density distribution in a universe with 512 nodes. As expected, larger views allow nodes to have more children thus reducing maximum depth. The larger depths in DAGs are because depth measures the maximum distance, i.e. the longest path from the root to the node, which increases with the extra number of links. The steep curves hint that the structures built by Brisa are fairly balanced, i.e., do not degenerate into long chain even with a simplistic strategy thus preserving desirable properties for dissemination. An analysis of the degree distribution confirms this observation.

The degree of a node in Brisa is given by the number of outgoing links and thus bounds the message copies a node sends. This is directly related to the dissemination effort and as such degree distribution should be as narrow as possible indicating an evenly distributed load. When analyzing the degree distribution presented in Figure 4.7 three main observations arise. First DAGs are more effective than trees in having a greater share of the nodes contribute to the dissemination effort (nodes with degree zero are leaves). As overall more links are required, the chance of having all outgoing links deactivated is smaller. Secondly, degree distribution is also highly affected by the view size provided by the PSS: higher values lead to shallower trees thus resulting in more leaves, while lower values lead to deeper trees due to the limitation imposed by the view sizes.

Such relation between degree and depth can be observed in Figure 4.8, which depicts sample trees obtained by Brisa. As a matter of fact, despite using a simple strategy, the

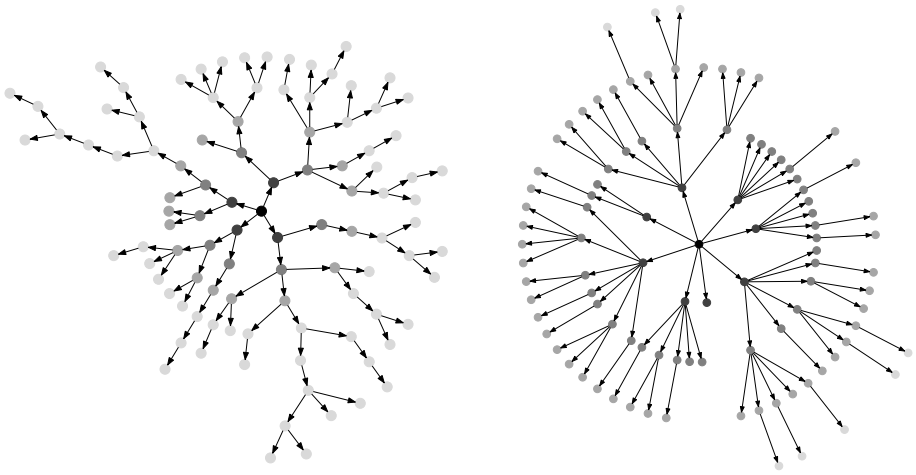


Figure 4.8: Sample tree shape for 100 nodes represented in a radial layout. The HyParView active view size of 4 (left) and 8 (right). Expansion factor is 1.

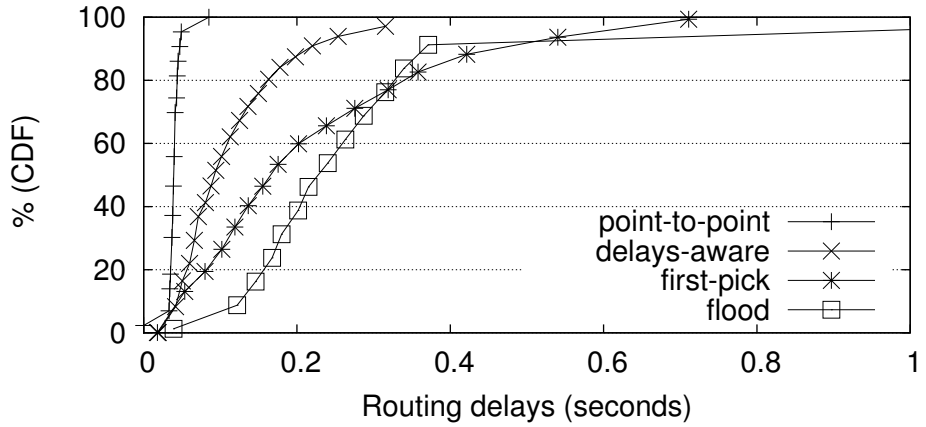


Figure 4.9: Routing delays distribution on PlanetLab for a 150 node network. Structure is a tree with view size 4. Message size is 1KB×200 messages.

resulting trees are fairly balanced which is essential for efficient dissemination. Finally, despite using an *expansion factor* of 2 the number of nodes with degree higher than the configured value (in our evalaution, 4 and 8) remains small as hinted in Section 4.2.1.

4.3.2 Network properties

In this section we focus on the network properties of the dissemination structures obtained by Brisa. First, we analyze the routing delay of dissemination. To this end, we use the cumulative round trip times, taken at each hop, from the root to a given node. When compared against the round trip time of direct communication between the root and that node, it indicates the effectiveness of Brisa in building dissemination structures with low end-to-end delays, an essential property for a dissemination system. The ratio between the first and second measurements gives the stretch factor. However, due to PlanetLab asymmetries that deter direct communication between some nodes, we instead present the

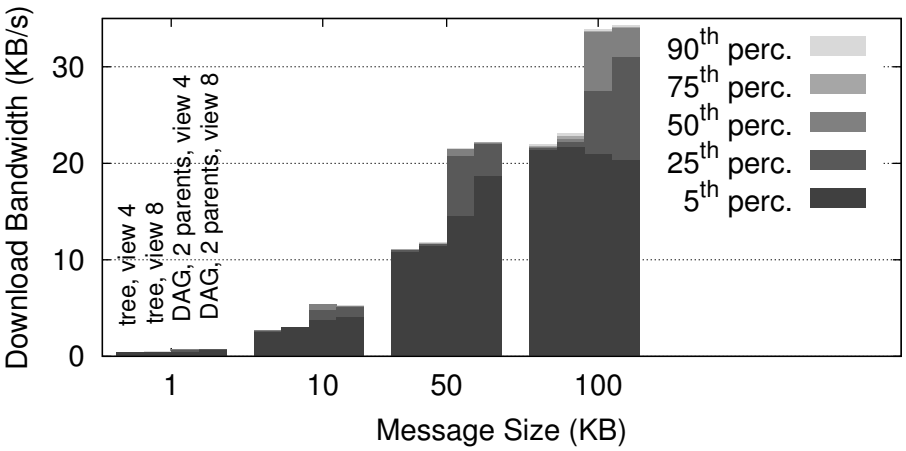


Figure 4.10: Bandwidth usage for a 512 nodes network, download.

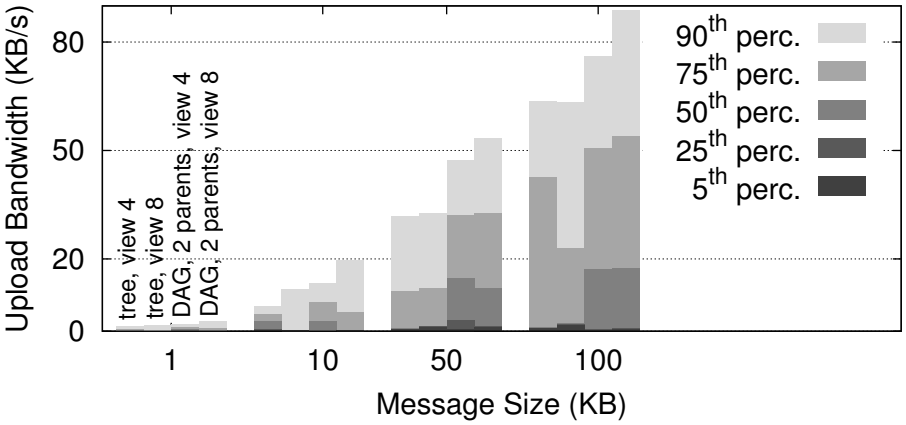
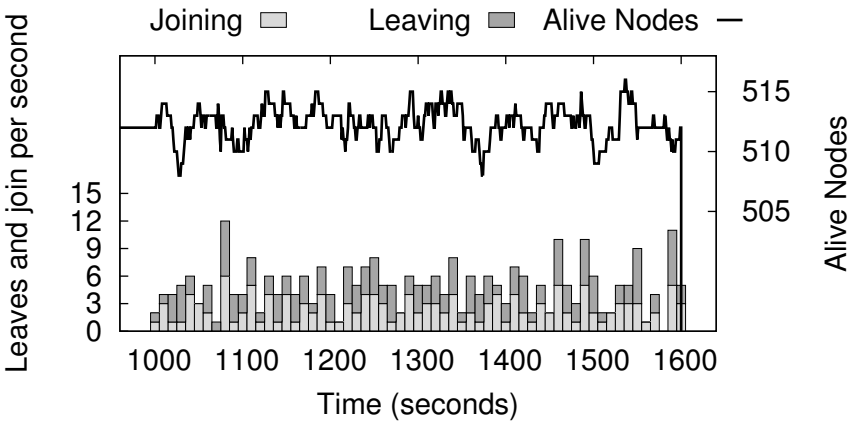


Figure 4.11: Bandwidth usage for a 512 node network, upload.

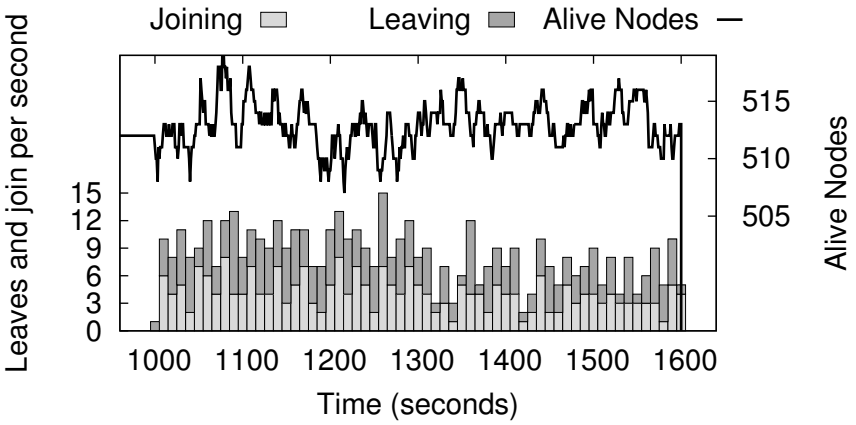
cumulative distribution of the raw results in Figure 4.9. Not surprisingly, the flooding strategy yields the worst results due to the heavy load imposed on the network. In this non-structural metric, the effects of a delay-aware strategy become clear when compared to the simplistic first-come first-pick: for instance, for 40% of the nodes the delay is reduced by 50%.

Next, we focus on the bandwidth usage. This measures the node's dissemination effort and is directly influenced by depth and degree distribution. Figure 4.10 and Figure 4.11 depict download and upload bandwidth usage, respectively, for payloads of 1, 10, 50 and 100 KB. We used stacked bars with decaying shades of grey for representing a distribution using a set of percentiles. For instance, the medium shade of grey gives the median value (half of the node below that value, the other half above), while the lighter shade gives the 90^{th} percentile: 90% of the nodes are associated with a lower bandwidth.

As expected, trees are more frugal with respect to download as nodes receive exactly one copy of each message whereas in DAGs nodes receive two copies (one for each parent). For each structure, the increase in bandwidth usage for the different view sizes is due to the PSS. The small difference, negligible when compared to application messages, hints at a low overhead service. The differences in the percentiles for the DAG are related to the depth of nodes (Figure 4.6) as nodes at lower depths may not be able to find additional parents and thus receive messages only from a single parent.



(a) $X = 3\%$ churn rate.



(b) $X = 5\%$ churn rate.

Figure 4.12: Synthetic churn trace, node dynamics (leaves/join) for $X = 3\%$ (top) and $X = 5\%$ (bottom) churn rate on a 512-nodes deployment.

For upload, results are naturally similar. DAGs require more links and consequently nodes will have to relay messages to more neighbors, increasing upload bandwidth usage. The differences between percentiles for a given configuration are explained by the degree distribution (Figure 4.7) as nodes with higher degrees need to upload more.

4.3.3 Robustness

We now focus on the behavior of Brisa under continuous churn in order to assess its robustness. Each experiment is associated with a synthetic churn trace based on the churn support module of Splay. The synthetic description is given in Listing 4.1 and proceeds as follows: first we bootstrap the system and let it stabilize. After, we induce churn at rate X by having X percent nodes fail at random and X percent new nodes join the system during each minute. Figure 4.12 depicts the resulting nodes dynamics induced by the synthetic churn trace in the case of 512 nodes.

```
1 from 1s to N s join N
2 at 1000s set replacement ratio to 100%
3 from 1000s to 1600s const churn X% each 60s
4 at 1600s stop
```

Listing 4.1: Churn trace generation script

Churn conditions		Parents lost/min.	Orphans/ min.	%Soft repairs	%Hard repairs
128 Nodes					
Churn rate:	Tree	2.3	2.3	87.0	13.0
X=3%	DAG, 2 parents	4.0	0.2	92.5	7.5
Churn rate:	Tree	3.4	3.4	79.4	20.6
X=5%	DAG, 2 parents	7.0	0.3	90.0	10.0
512 Nodes					
Churn rate:	Tree	22.2	22.2	88.2	11.8
X=3%	DAG, 2 parents	36.8	2.3	94	6
Churn rate:	Tree	22.2	22.2	87.7	12.3
X=5%	DAG, 2 parents	32.3	1.7	94.1	5.9

Table 4.1: Impact of churn for a 128 and 512 node networks with active view size 4.

Table 4.1 presents the results obtained for networks with 128 and 512 nodes. For simplicity we ensure that the source node does not fail. However, we note that the failure of source node would only produce a negligible impact in the presented results. In fact only the direct children of the source (a small number limited by the view size) would experience the effect of a parent failure. We defined the following metrics:

- **Parents lost per minute:** rate at which nodes lose any of their parents;
- **Orphans per minute:** rate at which nodes lose all parents (implying disconnections);
- **Percentage of soft repairs:** upon disconnections, how many nodes successfully repair their incoming links using the *soft repair* mechanism;
- **Percentage of hard repairs:** upon disconnections, how many nodes required using the *hard repair* mechanism.

As expected the rate at which parents are lost is higher for DAGs than trees due to the larger number of parents of the former. Nonetheless DAGs are much more robust with nodes being seldom fully disconnected. For instance, with a churn rate of 5% per minute, which implies half of the nodes leaving the system within the ten minutes of the experiment, only 17 nodes on an universe of 512 get disconnected (1.7 per minute * 10). Of those, all but one were able to recover using the soft repair, which simply implies activating a link to a new parent. Moreover, the time required for hard repairs, studied in the next section, is very low meaning that despite disconnections nodes are able to promptly repair connectivity with minimal effort. Finally, quick parent recovery also allows nodes to quickly recover lost messages thus ensuring that all application messages are effectively delivered. Such recovery capabilities under high churn, combined with efficient dissemination structures that are correct by design made Brisa a promising substrate for efficient and robust dissemination in the large scale (Chapter 5).

4.3.4 Support for multiple trees

In this section, we analyze Brisa's support for multiple trees regarding load balancing and performance. The network size is 512 and the active view size is 8 as for the previous experiments. Unless otherwise stated, the multiple tree experiments below do not use the tree reusing strategy; the goal is instead to create multiple, independent and disjoint trees.

We first analyze Brisa's multiple trees effectiveness in balancing the dissemination effort among all nodes. Figure 4.13 depicts the number of trees where nodes are leaves.

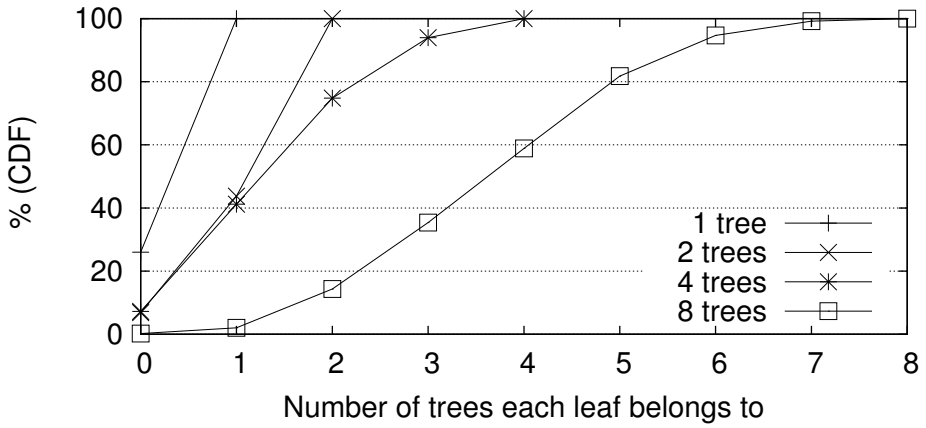


Figure 4.13: Distribution of the number of trees where nodes are leaves for a 512 node network with active view size of 8.

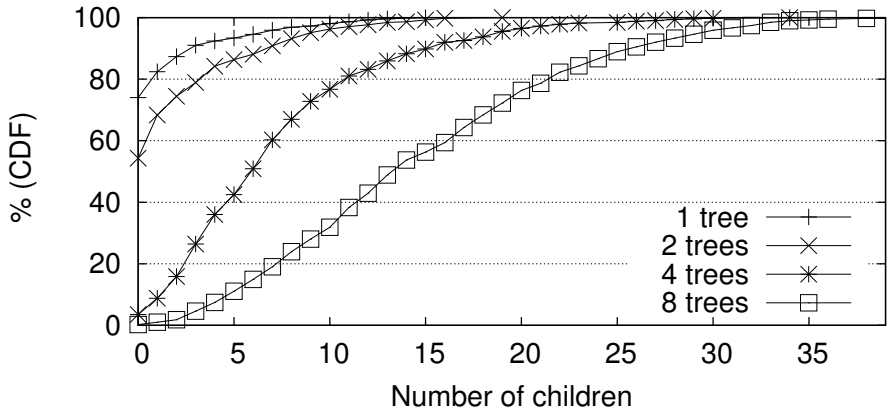


Figure 4.14: Distribution of the number of children across all trees for a 512 node network with active view size of 8.

For instance, with 2 trees, 40% of the nodes are leaves in one tree. Results confirm our expectations that as the number of trees increases, the chance of nodes being a leaf in all trees becomes dismayingly small, for instance for the 8 trees experiment only less than 5% of the nodes are leaves in more than 6 trees. As leaf-only nodes do not contribute to the dissemination effort, these results indicate that the use of multiple trees is essential to promote load balancing among nodes.

This is confirmed in Figure 4.14 which presents the number of children of each node across all trees. As is it possible to observe, the number of nodes that do not contribute to the dissemination, i.e. have zero children, diminishes dramatically with the number of trees in the system. In fact, with a single tree, almost 80% of the nodes do not upload whereas for 8 trees this value is very close to zero. These results confirm our motivation to use multiple trees as a mechanism to balance the dissemination effort among all the nodes (Section 4.2.8). We note that this is achieved without explicit coordination among nodes or by using more complex mechanism as in SplitStream [103]. In fact, Brisa just relies on the inherent randomness of the underlying PSS to build disjoint trees.

In the next experiment, we study the evolution of Brisa's performance with respect to the number of trees. This allows to access the impact deploying multiple trees has on the

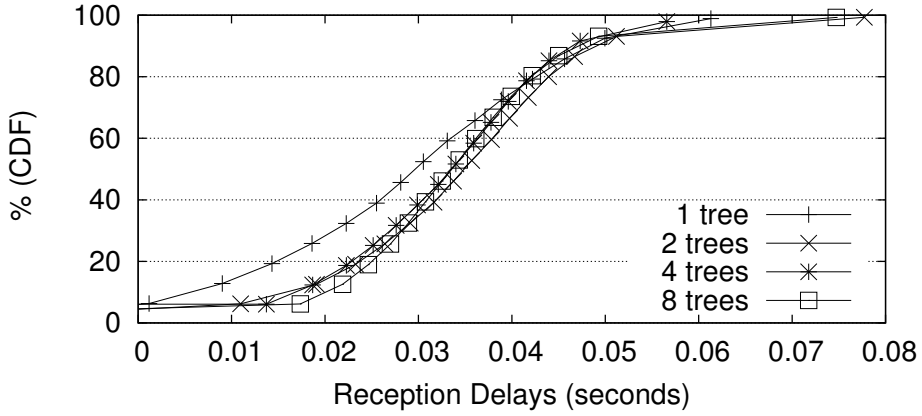


Figure 4.15: Reception delays per message when using multiple trees for a 512 node network with active view size of 8. The number of messages is 500.

reception delay of the individual trees. The reception delay is defined as the time elapsed, at the source, since the message was published until the reception at nodes, and gives the combined effect of: a) the routing delays inherent to the dissemination structure, and b) eventual delays due to the dissemination overhead (reception, processing and relaying of messages). Note that this measurement does not require synchronization among nodes: upon reception of a message, a node notifies the source which replies back with the time elapsed since the message was published. The resulting value is then weighted with the time elapsed since the node first sent the notification to minimize the network impact in the measurement. Results are depicted in Figure 4.15 and show that the reception delay is very similar regardless of the number of trees. This demonstrates that not only are Brisa's multiple trees effective in promoting load balancing among nodes but also the individual performance of multiple trees is similar to that of a single tree. We account for this behavior precisely due to the randomness in the tree creation process. As a matter of fact, as more trees are added, previously unused resources (leaf-only nodes' upload capacity), start being used enabling the performance of the system to remain stable despite the increased overall load.

Finally, we consider a scenario where multiple trees are used to split content and improve not only resource usage but also dissemination time. We note that this scenario is close to the one proposed in SplitStream where several disjoint trees are used to stream content.

In this experiment, we inject 500 messages on the system, evenly split across the given number of trees, and measure the dissemination delay. The dissemination delay is defined as the local time elapsed between the reception of the first message and the reception of all messages. Note that, while the reception delay measures the time elapsed since a message is published until it arrives at nodes, the dissemination delay measures the time it takes for a node to receive all messages. Results are shown in Figure 4.16. To improve readability, we show only the portion of the plot where the measurements lie. As expected, the dissemination delay is considerably reduced when increasing the number of trees. This is because more messages can be sent in parallel in each tree but also because the reception delay when using multiple trees does not increase. The cost is a naturally increased bandwidth usage due to parallelization. Such cost can however be observed in the distribution of children of each node, which essentially gives the upload requirement, enabling an application designer to choose the right amount of trees tolerated by the underlying physical network.

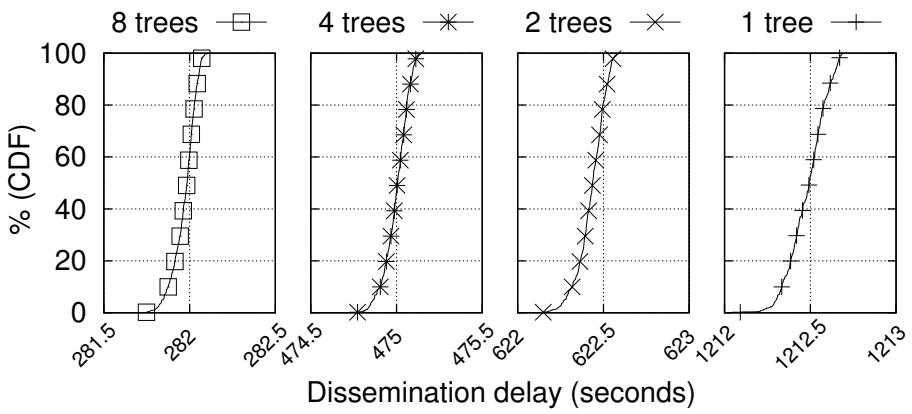


Figure 4.16: Dissemination delay when splitting the stream of messages across multiple trees for a 512 node network with active view size of 8. The number of messages is 500.

Protocol	**Organization**		**Dissemination Strategy**	
	Structured	*Unstructured*	*Push*	*Pull*
SimpleGossip	×	√	√	√
SimpleTree	√	×	√	×
TAG	√	√	×	√
Brisa	√	√	√	×

Table 4.2: Protocol design space.

4.3.5 Comparison with existing approaches

In this section we compare Brisa's bandwidth usage, structure construction time, dissemination latency and parent recovery delays with several alternative protocols. The protocols we compare Brisa with are representatives of different points in the efficiency/robustness design spectrum as can be observed in Table 4.2. The comparison metrics have been chosen because they generally represent the most important parts in any dissemination system and clearly show the impact of each design decision. For Brisa we use a tree with a HyparView active view size of 4. In order to assess the inherent overhead of each approach, and for fairness reasons, the other approaches are implemented and evaluated in the same environment as Brisa and configured with equivalent settings. The protocols we consider are the following:

SimpleGossip. This approach lies on the robustness end of the spectrum. We use Cyclon [31] as the PSS. Due to its proactive nature we use a combination of rumor mongering (push) to infect most of the nodes and anti-entropy (pull) to ensure completeness [112]. Rumor mongering follows an infect and die strategy with a fanout of $ln(N)$, where N is the system size and anti-entropy exchanges updates with a single random neighbor with a frequency that is the double of the message creation ratio.

SimpleTree. Oppositely, this approach lies on the efficiency side of the design spectrum. We consider a tree created randomly with the help of a centralized node. The only criteria for a node joining the tree is to connect to a parent that joined earlier in the past, which avoids creating a cycle in a similar manner to the one used in Tag. This parent is provided by the centralized node that randomly picks any of the previously joined nodes as a parent

for a newly joined node. Dissemination is done by pushing the messages immediately through tree links thus minimizing latency.

Tag. For this approach which tries to achieve both robustness and efficiency we use Tag [116]. As Brisa, Tag maintains a tree and an epidemic-based overlay to combine the efficiency of trees and robustness of epidemics. Nodes are further organized in a linked list sorted by joining time, with nodes maintaining information about their predecessors/successors up to two hops away. New nodes traverse this list backwards until an application specific condition is met. In the traversal, nodes pick k random peers to form the gossip overlay and join the tree by choosing a suitable parent. Upon parent failures, nodes update the linked list and traverse it again to find a new parent and thus restore the tree. Regarding dissemination, Tag uses a pull-based approach with nodes pulling content both from the tree and from overlay neighbors. Because Tag relies on pull we expect increased dissemination latency due to the additional roundtrips and pull period. We chose to compare Brisa against Tag due to its proximity in terms of goals and general approach (combining tree efficiency and gossip robustness) and the differences in its design choices (e.g., tree construction mechanism and a pull-based approach). We believe this choice allows a better assessment of the merits of each approach in the following evaluation scenarios.

Bandwidth usage. We first focus on the bandwidth usage of each protocol by considering two metrics: *stabilization bandwidth* and *dissemination bandwidth*.

Stabilization bandwidth is the bandwidth used to bootstrap the protocol including the construction of the overlay and tree structures and is measured until stabilization. After stabilization we consider the *dissemination bandwidth* as the bandwidth associated with message disseminations and subsequent management overhead. Once the structure stabilizes, we inject messages with payload sizes from 0 to 20 KB in a network of 512 node. This differentiation allows us to clearly observe the overhead imposed in each phase. As SimpleGossip does not uses any structure we represent all the bandwidth consumed under *dissemination bandwidth*.

Figure 4.17 presents bandwidth consumption averaged over all nodes. As expected, Tag and Brisa are comparable and the actual cost is dominated by the sending of data among peers rather than the management cost of bootstrapping the dissemination structures. The smaller management overhead of SimpleTree is due to the fact that only a single communication step with the centralized node is needed while the other protocols require inter-node communications. The small extra bandwidth cost for Tag and Brisa when compared to SimpleTree is from the maintenance of the PSS layer and dissemination structures that are key to the performance in terms of delays and robustness as we explore later. For the smaller message sizes, SimpleGossip is comparable with both Brisa and Tag due to the absence of structure management and because Cyclon does not uses explicit fault detection mechanisms. However, this is quickly offset for larger message sizes due to the excessive number of duplicates SimpleGossip relays resulting in high bandwidth consumption.

Structure Construction Time. In this experiment we measure the time necessary to bootstrap the dissemination structures both on the cluster and on PlanetLab. Due to the absence of structure of SimpleGossip and the construction simplicity of SimpleTree, they are not considered in this experiment. For Brisa we consider the time elapsed since a node sends the first deactivation message until all its inbound links except one are deactivated. In the case of Tag we use the time since a node joins the list until it settles its position on that

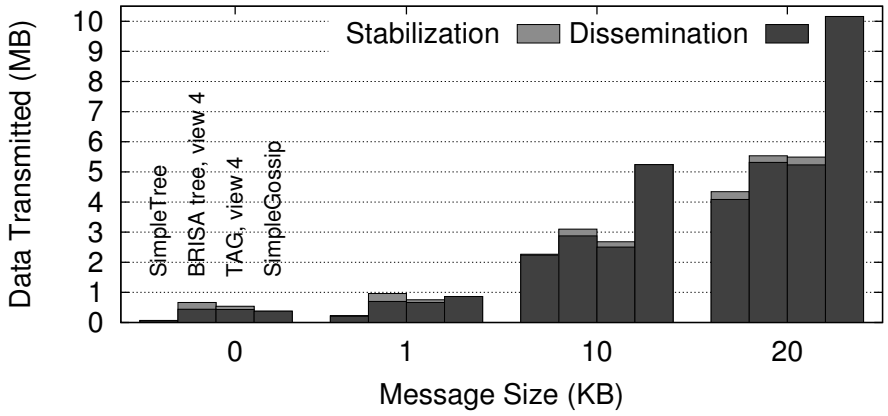


Figure 4.17: Bandwidth usage for a 512 node network.

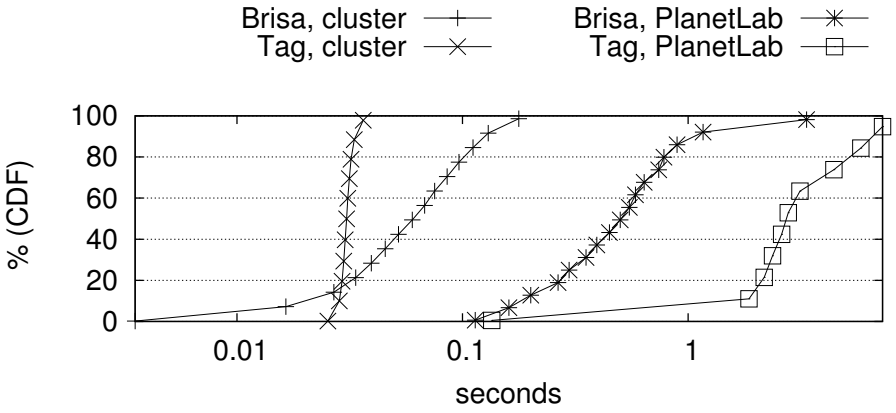


Figure 4.18: Construction time for 512 (on cluster) and 200 (PlanetLab) nodes.

list. Results are presented in Figure 4.18. It is interesting to observe that in absolute terms (note that the x scale is logarithmic) TAG is marginally faster than BRISA on the cluster but much slower on PlanetLab. This is because the construction mechanism happens at once in TAG by traversing the list, whereas in BRISA it is triggered by the reception of messages. As BRISA keeps the connection to its neighbors open, in the adverse environment of PlanetLab, the traversal cost of TAG (i.e. creating a connection to a node, exchanging messages, tearing it down and proceeding to the next node) easily outweighs the time BRISA needs to wait for the reception of the messages from all its neighbors.

Dissemination Latency. We consider dissemination latency as the time elapsed between the reception of the first and last message among the set of all messages. When studied

Protocol	**Latency (seconds)**	**Overhead**
SimpleGossip	128,23	+28%
SimpleTree	100,025	-
TAG	200,476	+100%
Brisa	106,587	+6%

Table 4.3: Dissemination latency for a 512 nodes network for 500 messages of 1KB.

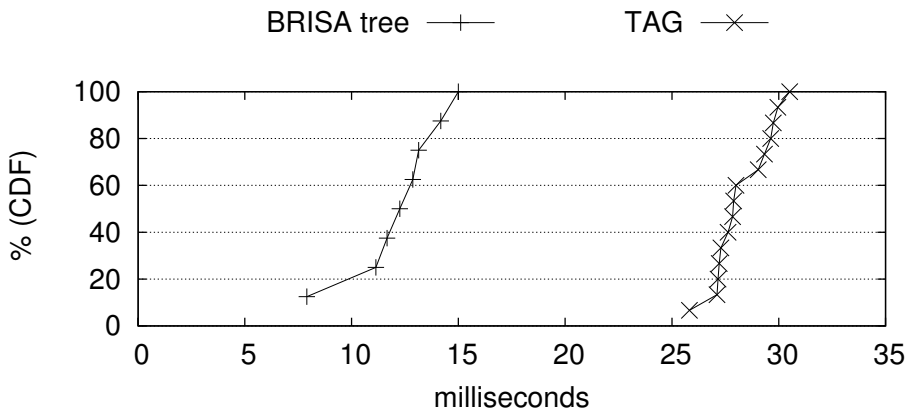


Figure 4.19: Parent recovery delays for a 128 node network with active view size 4, 3% continuous churn conditions.

along with bandwidth usage, it highlights the tradeoffs of each approach. The message payload is 1 KB and the the ideal dissemination latency is 100 seconds (500 messages at 5 per second). Table 4.3 presents the results averaged over all nodes. As SimpleTree is very close to the ideal value we use it as a baseline of comparison for the other approaches. Latency for Tag is significantly higher than the other approaches. This is mainly because Tag uses a pull-based approach to get updates, while the others rely on push. We note however that this is a characteristic that pertains to pull approaches in general and not Tag in particular. The delays for Brisa are similar to the ones for SimpleTree, with a small variation that we account for the extra context switching and physical machines sharing on our cluster. Differences in practice are expected to be minimal with SimpleTree, and largely in favor of Brisa when using a delay-aware selection strategy as previously illustrated in Figure 4.9. Somehow surprisingly, SimpleGossip performs worse than Brisa and SimpleTree. This is due to the overhead of dealing with duplicates and eventual omissions that need to be compensated by the slower anti-entropy mechanism.

Parent recovery delay. Our last comparison considers the robustness of Brisa and Tag. As SimpleTree does not consider dynamic scenarios, and SimpleGossip does not maintain any structure both approaches are ignored in this experiment. We apply for both protocols the same churn conditions as described in Section 4.3.3, with a churn rate of 3% and focus on the parent recovery delay for hard repairs in both cases. In Brisa this corresponds to the case where no immediate replacement neighbor is available and the underlying gossip layer is used. In Tag this corresponds to the case where the linked list is broken (i.e., two consecutive simultaneous node failures) and the node needs to be re-inserted into the structure. Figure 4.19 depicts the results in a 128 node network. We note that Brisa, while yielding a similar bandwidth cost, and better dissemination delays, also outperforms Tag regarding robustness in two ways: i) the number of hard repairs almost doubles with Tag (not shown) in the same churn conditions and ii) the delay for recovery is twice as fast for Brisa. This means that both the disruption of dissemination happens less often with Brisa, and that the effect of such disruptions is less than what is experienced with Tag.

4.4 Related Work

Existing approaches to large-scale data dissemination cover two main design domains: overlay management and application-level multicast. In the following we present existing work in this design space and compare it to our approach.

Scribe [108] is an application-level multicast layer that builds dissemination trees by aggregating reverse paths to a rendezvous node in the Pastry [121] DHT. Unlike Brisa, where we assume that all nodes are interested in all messages, Scribe supports group membership management by having each node subscribing to group(s) it is interested in. Yet, the load of dissemination is shared by non-members of the groups that must act as interior (forwarding) nodes in the dissemination trees. Unlike epidemic-based dissemination, where the failure of a node has little impact on the system, Scribe's *rendezvous* nodes are single points of failure and bottlenecks in the system. Brisa also constructs a dissemination structure from an existing overlay, but can leverage the epidemic dissemination layer as a fallback for robustness. We note that group membership can be implemented in Brisa by maintaining on each node separate views for its subscribed groups, as done in the TERA publish/subscribe system [122]. These group specific views can themselves be constructed by the means of an epidemic-based clustering protocol [24].

SplitStream [103] is a high-bandwidth dissemination layer built on top of Scribe [108] and Pastry [121]. In order to balance the load of dissemination, it constructs multiple Scribe trees that are used for sending alternate pieces of a stream; nodes that participate as a leaf in one tree participate as an interior node in the other(s), thus balancing the in- and out-degrees of nodes. The same is achieved probabilistic by Brisa due to the inherent randomness of the PSS where the multiple Brisa trees are embedded.

Chunkyspread [123] also builds multiple dissemination trees, rooted at a single source node. These trees are built on top of an unstructured overlay and not on a DHT. They are used to parallelize the dissemination process by pushing different parts of the data in each tree. Cycles in the trees are avoided by using a technique derived from Bloom filters, whereas Brisa relies on simpler mechanism based on the path or the number of hops from the source. Chunkyspread trees can be constructed by taking into account latency and load metrics that can also be considered with Brisa's parent selection strategies.

In Bullet [124], a stream of data is also pushed through a tree structure. Different data blocks are intentionally disseminated to different branches of the tree, taking into account the bandwidth limits of participating nodes. Bullet complements this tree with an epidemic-like layer that allows the recovery of missed messages. This mechanism takes the form of a mesh that is used to locate peers with missing items, in a way similar to a PSS. In this sense, Bullet is based on a design choice that is opposite to ours: Brisa complements a robust dissemination layer (the PSS) with an efficient but failure-prone structure (tree/DAG), while Bullet complements a tree with an epidemic-style dissemination to support failures. Rappel [125] is another example of a dissemination service that combines a tree structure for dissemination with an epidemic-based service for optimization. In the case of Rappel, the epidemic-based layer is used to locate suitable peers based on interest-affinity and network distances, and not as a fallback mechanism for dissemination.

MON [109] relies on a mechanism similar to Brisa to construct spanning trees and DAGs on top of an unstructured overlay. The goal of MON is to manage large-scale infrastructures such as PlanetLab, by using the resulting trees/DAGs to disseminate management commands. Therefore, sessions in MON are intended to be short-lived and the protocol does not provide any support for dynamism in the population of peers. To disseminate

data, MON relies on a pull strategy, where nodes can download content simultaneously from multiple parents, if available. This approach eliminates duplicates, as it is the receiver that decides which pieces to receive. However it requires nodes to maintain knowledge of the data blocks/messages present at each parent.

The work presented in [126] stems from an observation similar to ours that even though epidemic-based dissemination is attractive due to robustness, achieving completeness requires large fanouts resulting in high overhead. The authors thus propose a hybrid approach that uses an epidemic-based dissemination with fanouts lower enough to infect most of the population, and ensures completeness by relying on a ring structure that encompasses all nodes. Epidemics are used for the bulk dissemination of data, still resulting in many duplicates, as opposed to Brisa, where most of the dissemination happens on the dissemination structure with a controlled number of duplicates. Similarly, in [127, 128] a Chord-like ring overlay is combined with a push mechanism to disseminate messages over a spanning tree optimized for minimal latency. Brisa instead builds on top of an unstructured overlay, and it offers a wider set of options for the tree construction.

In [129] the authors propose an alternative approach to tree repair based on proactive principles. Each node computes alternative parents for its children that can be used upon failures. This minimizes disruptions as nodes know beforehand the new parent they need to connect to. Further it can cope to some extent with multiple concurrent failures and strictly control node degrees, a major goal of the authors. Due to this restriction, tree shape tends to degenerate to a chain overtime penalizing end-to-end delays. Brisa uses a notion similar to the alternative parents without however having the tree degenerate into a chain. This is because [129] only considers potential parents in the failed node subtree while Brisa can consider any node as long as it passes the cycle detection mechanism.

GoCast [130] builds a dissemination tree embedded on an epidemic-based overlay that takes into account network proximity to improve end-to-end latency. The tree is built using a traditional Distance Vector Multicast Routing Protocol (DVMRP) and used to push messages as in Brisa. Message identifiers are advertised through the overlay links as in PlumTree [131]. Such identifiers are used to recover missing messages, for example due to tree disruptions. Contrary to Brisa, these identifiers require additional network overhead. Most strikingly this recovery information is not used to repair the tree, which relies solely on DVMRP and thus presents scalability problems due to the overhead of periodic floods to rebuild the tree. Furthermore, Brisa is able to adjust to different performance criteria but could nonetheless take advantage of the network-proximity offered by Gocast's overlay. TAG, the protocol we use in the direct comparison with Brisa also falls into this class due to the use of a tree and an epidemic-based overlay. More details can be found in Section 4.3.5. PlumTree [131] also relies on the detection of duplicates and subsequent deactivation of links to build an embedded spanning tree on an underlying unstructured overlay. However, inactive links are still used in a "lazy push" approach, by announcing the message identifier instead of the full payload. These announcements are used to repair the tree: when an announcement for an unknown message is received, the protocol starts a timer. If the timer expires before the reception of the payload the tree repair mechanism is triggered. This approach is highly sensitive to variations in network latency, which leads to unnecessary message recoveries as observed in [132]. Brisa does not separate the dissemination of the header and payload, the dissemination is deterministic, and faults are detected thanks to the underlying PSS layer, which avoids sending periodic probe messages at the level of the dissemination layer. Further, the generic construction mechanism can build trees and DAGS according to different criteria, which is not possible in PlumTree. Due to the use of message advertisements to manage

faults both PlumTree and GoCast fall in an undesirable tradeoff: either advertisements are sent sparingly to conserve bandwidth with an impact on recovery time, or advertisements are eagerly sent imposing a constant management overhead to the system.

Thicket [132] uses the same principles of PlumTree to build multiple dissemination trees on top of an unstructured overlay. The goal is to provide similar functionality to SplitStream by balancing the number of trees where a node is interior and also by splitting the content among trees to improve fault-tolerance. The mechanism used to build trees imposes several constrains that do not ensure the resulting tree is connected by design. This is addressed with a tree repair mechanism based on missing messages, as in PlumTree, that requires periodic exchanges of received messages among neighbors which is also used to handle joins and leaves. The support for multiple trees in Thicket is based on the premise of load balancing and fault-tolerance by leveraging on network coding techniques. The cost however is a linear growth in the number of links with respect to the number of trees, which poses scalability concerns. In contrast, Brisa builds connected trees by design, despite controlled fanouts, and deals with joins and failures with a simple and lightweight mechanism that is triggered only when failures happen. Multiple trees are a natural extension of the system and therefore do not require additional maintenance mechanisms.

4.5 Summary

In this chapter we presented the design and evaluation of Brisa, a data dissemination system that combines the robustness of epidemic-based protocols and the efficiency of structured overlays. Brisa automatically emerges efficient dissemination structures from the flooding-based distribution of the first message in a stream. The construction of efficient dissemination structures exploits the path diversity that naturally exists in epidemic- and flooding-based dissemination, while avoiding the high level of duplicate reception these mechanisms typically yield. Robustness comes from the ability of the underlying epidemic layer to rapidly provide replacement nodes upon failures, and by acting as a dissemination fallback. Therefore, Brisa bridges the gap between robust but costly epidemic or gossip-based dissemination and efficient but failure prone structured approaches. We evaluated Brisa with a prototype deployed on a cluster and on PlanetLab. The experiments and comparisons to related work confirmed Brisa as a robust and efficient system for data-intensive applications.

Chapter 5

LayStream: A Layered Approach to Gossip-based Live Streaming

This chapter presents LayStream, a system that leverages a stacked composition of gossip-based protocols to implement a complex application, in this case video streaming. We report on the lessons learnt in building such an application, in the context of a large-scale deployment. Its implementation partly integrates the dissemination mechanisms contributed by Brisa in the previous chapter. LayStream contributes a topology construction layer that builds spanning graphs on top of geographically distant nodes. This topology-awareness feature of LayStream consolidates its sound results in the context of large-scale networks. Gossip protocols reside at the core of each of layers composing the LayStream stack. This chapter documents our experiences in building and combining such gossip-based layers.

5.1 Introduction

Originally introduced more than twenty years ago for propagating updates in distributed databases [112], epidemic or gossip-based protocols have attracted increasing interest as the scale and dynamism of computer systems have drastically increased over the last decade. A wide variety of gossip-based protocols have been proposed, for purposes such as fault detection [28], membership management [31], aggregation [20], data dissemination [110, 133], publish-subscribe [122, 134], or infrastructure management [135], among others. This diversity stems not only from the robustness and scalability of the gossip model, but also from its remarkably simple principles. A peer participating to a gossip protocol follows a simple periodic interaction pattern: it selects a communication partner and exchanges some data with it, allowing each of the two peers to update a local state according to protocol-specific rules. Due to the randomized nature of the interactions between peers, gossip protocols are very robust against faults and churn, and they are therefore well suited to dynamic environments. As each peer only needs limited knowledge of the entire system, typically restricted to a few dozen other peers, gossip protocols can scale well to very large numbers of peers. This intrinsic characteristic is perfectly aligned with the requirements of modern global systems and applications. Finally, the ability of the peers to autonomously organize themselves in a completely distributed manner, a property known as self-organization, further extends the robustness and scalability of gossip protocols as no central management authority is required.

Despite being simple from a single peer perspective, the emergent and dynamic nature of gossip protocols makes it hard to understand, model, and evaluate their behavior on a global scale. Furthermore, while one can relatively easily compose gossip protocols to build richer

distributed services [122, 136], the resulting gossip stack of protocols is not straightforward to study and reason about. There is not only a lack of evaluation of individual gossip-based protocols, but their interplay in real environments remains largely unstudied. The mismatch between the behavior predicted by theory or simulations and results obtained in practical settings has been pointed out before, further highlighting the challenges of deploying key protocols such as consensus in the real world [137, 138].

In this chapter, we report our experience of building and deploying a composition of gossip-based protocols. Our use case is a live video streaming application with strict requirements on performance and robustness. The implementation, called LayStream, relies on the following components: a peer sampling service [30], a topology construction service [24], and a dissemination service built atop a tree construction and maintenance protocol [133]. These components are stacked and rely on the guarantees of the underlying layers , thus ensuring simplicity and isolation of concerns. We rely on the VLC/VideoLan[1] media player as the source and sink of video streams. A VLC endpoint injects a live stream that is disseminated by LayStream to multiple peers and played back by VLC.

Contributions

We developed and deployed LayStream using the Splay framework presented in Chapter 1.3.1. We report on the algorithmic and implementation challenges encountered for each component, and we evaluate them both independently in isolation and collectively in the final service. The chapter unearths some issues typically overlooked by previous studies, mostly due to the simplified nature of the simulations used for their evaluation. We propose solutions or alternative approaches to address each of these issues.

Outline

The rest of this chapter is organized as follows. Section 5.2 describes the protocol stack and the requirements that each layer has on the underlying ones. Section 5.3 presents the implementation of the stack and reports our experience at adapting existing protocols to match the requirements of the composition. Section 5.4 presents the evaluation of our prototype in a real deployment. Section 5.5 reviews related work and Section 5.6 concludes.

[1] `https://videolan.org/vlc/`

5.2 Live Streaming Components and Requirements

We start by describing the stack of services required to support streaming live video to a large set of peers in a decentralized and autonomous way. We follow a bottom-up description and focus on the abstraction and guarantees that each of the layer provides to the upper layers, up to the application (VLC). Implementations of each of the services are detailed in the next section. Figure 5.1 presents the stack and dependencies between the different layers. The base layer is a point-to-point wired communication layer. We assume that any two peers can directly communicate with one another. While this does not hold in general due to the presence of NATs and firewalls, techniques exist at the level of the first layer of the stack (peer sampling) for addressing this restriction (see Section 2.2.2 about the *Nylon* protocol).

The first layer provides a peer sampling service (PSS). A generic framework to implement PSS was introduced in Section 1.3.2. We evalute several PSS configurations in the context of the generic frawork. Details of this study are given in Section 5.3.1.

The second layer provides a *topology construction* service [24, 131, 139]. Its requirements depend on the model considered by the third layer, the *dissemination* service [131, 133]. The first requirement is to maintain a stable and bi-directional graph between peers. Each peer is equipped with a view, maintained separately from the one of the peer sampling layer but containing peers initially obtained from that service. The topology construction service maintains persistent bidirectional TCP connections (Section 4.2.3) towards all peers in this view. It also acts as a failure detector, using persistent connections to detect unresponsive peers and replacing them by live ones obtained from the constant stream of peers produced by the peer sampling service. The graph formed by these persistent connections has the important requirement that it must remain connected. Note that the underlying peer sampling graph is naturally connected with high probability thanks to its random graph nature, which ensures that the overlying dissemination layer can always reach all peers in the network. One possibility to ensure connectivity of the dissemination graphs would be to leverage randomness in the same way, and randomly pick a subset of the peers provided by the peer sampling layer as in [114]. This is however inappropriate for our target application, live video streaming, which is bandwidth-intensive and requires low transmission delays.

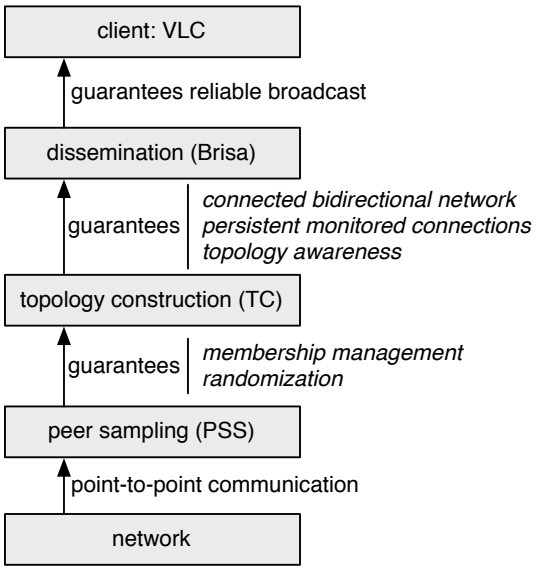


Figure 5.1: Services and guarantees.

Indeed, as packets will be transmitted over multiple links between the source and each of the destination peers, using arbitrary links would create unnecessarily long delays and pressure on the underlying network. Therefore, an additional goal of the topology construction layer is to link peers according to a proximity metric, e.g., in terms of delay or geographical locality. It does so by gradually selecting *closer* peers according to the considered distance metric. Note that this biased selection policy is an *optimization* that must not break network connectivity, which is a *strict* requirement of the dissemination layer.

The third layer provides a *dissemination* service (Chapter 4). Its design is remarkably straightforward thanks to the guarantees provided by the underlying layers. A simple flooding strategy is guaranteed to deliver disseminated messages to all peers, because of the connectivity and bidirectional nature of the underlying graph. Note that this is not the case for epidemic dissemination protocols directly based on the peer sampling service [110], where a periodic pull mechanism is additionally required to ensure complete coverage. Flooding is robust but particularly inefficient as it yields many duplicates. The role of the dissemination layer is thus to construct efficient dissemination structures (namely, trees) and maintain them in face of faults reported by the topology construction layer.

On top of the stack lies the application itself, in our case the VLC media streaming player. It only interacts with the dissemination layer and receives video packets as if they were originating directly from the source via a point-to-point UDP connection. We maintain a buffer of received packets and start the VLC client only when a configurable number of packets is present in the buffer, to accommodate fluctuations in reception delays due to churn and ensure a smooth video display.

5.3 Gossip-based Building Blocks

We describe in this section the implementation of the three services that compose LayStream and discuss the associated design decisions.

5.3.1 Peer Sampling

The peer sampling service layer is implemented using the gossip framework described in [30] and summarized in Chapter 1.3.2. This is the framework that allows to instantiate peer sampling services such as Cyclon [31] or Newscast [117]. Each peer in the network maintains as its *view* a list of c references to other peers. A peer is designated by a descriptor containing a `ip`:`port` pair and an `age` field indicating its freshness. The view is initially filled by operating a random walk in the existing graph. Each peer p updates its view by means of *view exchanges*, initiated by an active task on one peer and served by a passive task on another peer. The implementation of the peer sampling service faces an inherent trade-off between its two objectives of randomness quality and membership management which is controlled by parameters H and S discussed below. The active task, called periodically and at the same frequency on each peer, first selects a target peer q and constructs a list with $\frac{c}{2} - 1$ descriptors randomly selected among the $c - H$ newest entries, as well as the identifier of p with age 0. The list is then sent to q, which replies back with a sample of its own descriptors chosen in the same way. Node p integrates the received descriptors in its own view. To keep the view size constant to c elements, p first drops the H oldest elements and, as needed, also removes the first S descriptors sent to q, and then random descriptors. Node q proceeds similarly. Finally, both p and q increment the `age` of all descriptors in their views.

The *healing* and *swapping* parameters H and S, with the constraint that $H + S \leq \frac{c}{2}$, control the trade-off between randomness and membership management. The randomness quality is measured with the *in-degree distribution* and the *clustering factor*. The in-degree denotes the number of occurrences of a peer in the views of the other peers. A balanced distribution of in-degrees yields good load balancing. The clustering factor indicates that the peers in a view are also neighbors themselves. A random graph exhibits low clustering. A high value would make the graph vulnerable to massive failures and churn, and impede convergence for the protocols using the peer sampling service. On the side of membership management, the goal is to ensure that stale peers get removed from other views as fast as possible (in terms of number of exchanges, hence in term of the age of the corresponding descriptors). A value of $H = \frac{c}{2}$ (as in Cyclon [31]) favors randomness quality, at the price of a slow discarding of stale peers, while a value of $S = \frac{c}{2}$ (as in Newscast [117]) favors the quick removal of stale peers but leads to higher clustering and unbalanced in-degrees.

Atomic Exchanges

When implementing the PSS layer, we faced an issue that does not arise in simulations. The passive task at peer p can reply to requests at virtually any time. If an ongoing exchange has been initiated by p and is being served by another peer q, the modification of the view performed by p's passive task will break the exchange semantics. It might happen, for instance, that a descriptor gets duplicated or dropped. This scenario can also lead to artificial clustering. Such situations should be avoided: while concurrent modifications seldom happen, they introduce shifts that persist throughout the lifetime of the system and worsen over time. For instance, a descriptor that gets duplicated results in an increased in-degree for the linked peer, which leads in a higher chance of being contacted by others, and in turn in a higher chance of being subject to that shift again. A simple solution is to make the exchange atomic by locking the access to the view while the active task is pending. This requires careful design in the presence of churn and the use of appropriate timeouts. An alternative solution is to simply reject incoming requests at the passive task when an exchange is pending as done in [140]. We opted for the first solution in our implementation.

5.3.2 Topology Construction

The topology construction layer is implemented using an evolution of the T-Man protocol [24]. The goal of this layer is to create a graph (overlay) among peers that matches the structural needs of the overlying dissemination layer: links must be bidirectional and the network must be connected. Additionally, peers should be linked to close peers in terms of delays and geographical proximity, for reasons of performance and network utilization.

The T-Man Protocol

The basic operation of T-Man is similar to that of the peer sampling protocol. However, the selection of descriptors that are kept in the view of each peer after the exchange does not obey random decisions but depends on some semantic information included in the descriptor. In our case, this semantics is the 2-dimensional geographical coordinate (latitude and longitude) of the peer. We gathered the coordinates of a set of 200 PlanetLab peers for our evaluation[2]

[2]Since PlanetLab machines can be co-hosted, we prevent ties by introducing a small amount of noise to the obtained coordinates.

Figure 5.2: Geographical distribution of peers.

using a *GeoIP* service.[3] The selection of which descriptors to keep is based on a *distance* function. Each peer sorts all other descriptors according to this distance, with the objective to keep a view composed of the c *closest* ones.

Since there is no global knowledge of the peers, the topology self-organizes gradually as follows. Periodically, the active task at peer p selects another peer q from its view, and sends it its own view (or the best subset according to q). The passive task at peer q also returns q's current view. Both p and q merge their views with the set of received descriptors, sort it according to the distance function, and keep the c closest entries. Furthermore, on bootstrap and periodically after that, an exchange is initiated with a random peer obtained from the underlying peer sampling service. This process guarantees that the graph will converge, but it may take a long time when using a carelessly designed distance function (e.g., if each peer needs to encounter each other peer in its peer sampling view). A requirement for fast convergence is that the distance function be transitive: if peer p is close to peer q, and peer q is close to peer r, then p is also close to r. Our distance function is based on Euclidian distance between geographical coordinates and is therefore transitive. One trivial modification we also add to the original T-Man is to make all links bidirectional.

Our initial attempt to topology construction used the standard T-Man protocol applied to network coordinates from Vivaldi [141]. In a nutshell, Vivaldi maps the physical location of peers into a synthetic coordinate space. Each peer is associated to a point in that space and distances between points reflect the latency between peers. The coordinates of the peers are updated based on delays observed on application-level messages and is completely decentralized, thus seemingly well adapted to our context. At each peer, we simply chose the c closest peers according to the Euclidean distance between their respective coordinates.

We quickly noticed, however, that such a simple approach does not match our requirement of a connected topology (Section 5.2), in particular with our PlanetLab dataset. This dataset, represented by Figure 5.2, is clustered with one large group in the US, another one in Europe, etc. As a result, peers select as neighbors other peers in the same cluster and the resulting graph is partitioned. A trivial attempt to solve this problem is to add a set of additional, randomly selected entries in the view, but this approach proves unsatisfactory. Indeed, using random links introduces delay penalties that are propagated to all peers, and

[3] `http://www.maxmind.com/en/geolocation_landing`

keeping the network connected, in particular under churn, requires a large number of such links, clearly diminishing the interest of using delay-based peer selection.

Constructing a Spanning Graph

Achieving the apparently conflicting goals of connectivity and low distance between peers requires a careful adaptation of the T-Man protocol in order to guarantee system-wide structural properties that are not achieved by simply selecting the c closest peers. Connectivity in a network with bidirectional links is equivalent to the ability of reaching all peers from any peer of the network, e.g., by building a *spanning graph*. A spanning graph contains an embedded *minimum spanning tree* made of all the vertices and a subset of edges that guarantee connectivity but minimizes the sum of the associated costs. In our case, the cost for an edge is the geographical distance between two peers and the spanning graph is a subgraph of the complete graph exposed by the continuous stream of peers provided by the peer sampling layer.

Several approaches have been proposed for constructing spanning graphs, some of which can be implemented using gossip protocols. Typical examples are Delaunay triangulations such as [142, 143], but their main drawback is that the in-degree of peers is unbounded. Since we require bidirectional links, a peer may end up maintaining a very large number of TCP connections. Instead, we took inspiration from ad-hoc wireless networks, where spanning graphs are necessary to establish routing protocols and distances in the graph directly refer to wireless power requirements and energy consumption—two elements that must clearly be minimized. Wang and Li introduced Yao-Yao graphs in this context [144]. Yao graphs [145] are defined on a 2-dimensional Euclidean space (the space of geographical coordinates in our case). Each peer n is associated with k equally-separated rays, originating at n, that define k *cones*. In each cone, n selects the closest peer and connects to it with a directed edge. It has been shown that the Yao graph is a spanner for $k \geq 4$ [146]. Note that the Yao graph has a bounded out-degree k but the in-degree is unbounded.

The Yao-Yao [144] graph builds on top of the Yao graph. For each cone, we discard all incoming links but the shortest one, and make this link bidirectional. Note that this might result in some peers having no neighbor in one of their cones, which does not impact on the properties of the structure. The degree is thus bounded by k both for incoming and outgoing links, which will ensure good balancing of the load. The Yao-Yao graph is also a spanner [147].

To construct the Yao-Yao graph, T-Man must be modified to consider a separate entry in the view for each of the k cones.[4] All entries are bootstrapped using descriptors from the peer sampling service if they are located in the corresponding cone. Descriptors received by means of the gossip exchanges are considered only for the cone they lie in. The previous entry is replaced only if the received descriptor designates a closer peer. Figure 5.3 presents an example of evolution of the view of a peer p based on gossip exchanges with its neighbors. While the illustration uses $k = 6$ for the sake of clarity, we use $k = 8$ in our implementation.

Resource Utilization

The T-Man protocol was evaluated in [24] by simulations only. When deploying the protocol on real machines, we faced the problem of network resources limitations. The view is highly

[4]Such a modification to T-Man is also introduced by the authors of T-Chord [27], a self-organizing version of the Chord DHT where each entry of the routing table is maintained independently using its own distance function.

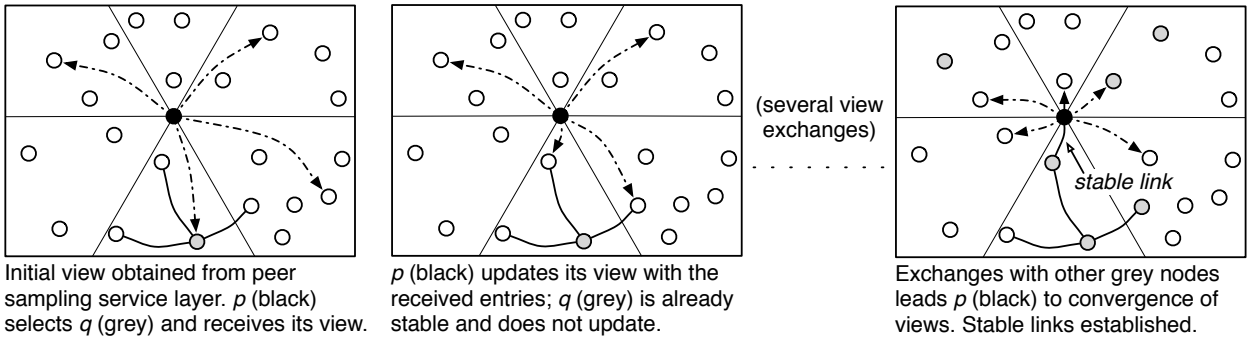


Figure 5.3: Evolution of the view of peer p towards Yao links for $k = 6$.

dynamic, in particular after a peer joins the system and until it evolves towards its stable configuration. Immediately establishing TCP connections when peers are added to the local view is problematic both for resource utilization and load balancing, in particular because many of these connections will be short lived and discarded at the next gossip exchange. TCP consumes more resources than UDP and requires maintaining connections on both ends of the links (e.g., file descriptors). Some peers might also be at the boundaries of clusters and pass through the views of many other peers. Therefore, establishing temporary bidirectional communications with all these passing-by peers is a clear waste of resources. In extreme cases, this can lead to resource exhaustion on one side or the other.

We solve this issue by using UDP for all gossip exchanges. We only establish TCP connections when the corresponding entry in the view has stabilized, which, based on experimental observations, we assume to happen after 5 consecutive gossip exchanges without modification. Note also that we only consider discarding existing bidirectional links (due to the Yao-Yao construction rule that allows at most a single link per cone) when we establish the persistent TCP connection to the new peer. Because of this only stabilized links are exposed to the upper layer.

5.3.3 Efficient Broadcast: Dissemination Layer

The overlay created by the topology construction layer is a spanning graph with bidirectional links. It follows that a flooding operation (where each peer forwards the first occurrence of an incoming messages from a peer to all its other peers) is guaranteed to reach all peers. Such a mechanism is obviously highly inefficient and not adapted to bandwidth-intensive video streaming. Each peer receives each message from multiple peers, up to c in the worst case. The dissemination is nonetheless extremely robust, thanks to these duplicates and in particular to the multiple alternative paths for receiving packets they correspond to.

The goal of the third layer is to provide an efficient *Dissemination* service. This service essentially constructs efficient dissemination trees embedded in the spanning graph provided by the underlying topology construction service. It inherits the robustness of its spanning graph. The links that are not currently active as part of a dissemination tree can be rapidly used as backup links upon failures—as these are maintained and monitored as persistent TCP connections. We use a subset of the Brisa gossip protocol (Chapter 4) for the implementation of this service. PlumTree [131] follows a similar approach and could be used as an alternative implementation.

Initially, all links exposed by the topology construction service are *active*. Upon receiving a message from a neighbor, peer p propagates it to all active links. The dissemination

of the first message thus corresponds to flooding. Thereafter, a deactivation mechanism allows selecting a single parent for each peer, effectively forming a tree: if a peer p receives the first copy of a message from q, it simply deactivates all links but the one from q. Note that deactivating the link is only performed at the level of the dissemination layer: this link is still maintained as a persistent TCP connection at the topology construction layer. The failure of the parent peer will be detected by this underlying layer and trigger an up-call to the dissemination layer. The peer then simply re-activates all its links and selects as a parent the first peer that sends the next packet. Albeit deceptively simple, this mechanism, based on local and simple decisions, is correct (Section 4.2.2). The resulting tree is a spanning tree (it covers all peers) and has no cycles (Section 4.2.4).

Note that we do not try to build the unique *minimal* spanning tree. The first reason is that the rigidity of the parent selection (the minimal spanning tree has, by definition, a single best parent for each peer) is contradictory with our robustness goal, and a reconfiguration would potentially trigger parent changes for other peers as well. Second, since the underlying spanning graph only contains low-delay links, the resulting tree has characteristics that are already very close to those of the minimal tree constructed using complete knowledge of the system.

The use of a single tree is efficient in terms of duplicates but results in high load differences between the peers. Some peers may contribute a high upload bandwidth while others are only leaves and do not contribute. We address this issue by constructing several trees concurrently using the same activation/deactivation mechanism. The only change needed is for control messages to carry an identifier that specifies the tree to which they belong. The source peer then splits the video packets onto the available trees, allowing for parallelization of the data dissemination and a more balanced distribution of load, similarly to Splitstream [103].

5.4 Evaluation

We focus on validating the service guarantees each provide to the upper layer. Then, we evaluate the performance of the complete stack under a live video dissemination workload. All our experiments were conducted using Splay [15], a framework for the implementation, deployment and evaluation of distributed systems.

We use a cluster of 14 bi-quad-core Xeon machines (112 cores in total, 224 with SMT), each with 8 GB of RAM and interconnected using a switched 1 Gbps network. We deploy 200 peers on this cluster.

5.4.1 Peer Sampling Layer

We start by evaluating the two fundamental guarantees of the peer sampling layer: the construction of a random overlay and membership management. The former is measured in terms of clustering degree and in-degree distribution, and the resulting balance in bandwidth utilization. The latter is measured as the time required to remove stale entries from the peers' view after a failure. Our experiments reproduce some of the results obtained by simulation in [30, 31, 117, 139] but using a real implementation.

We instantiate four protocols in the framework of [30] (see Section 5.3.1). The view size is always $c = 20$. *Blind*, with $H = 0$ and $S = 0$ essentially performs random selections for view selections at each exchange. *Swapper* corresponds to Cyclon [31] and uses $H = 5$ and $S = 0$. It favors randomness quality and in particular seeks at reducing clustering

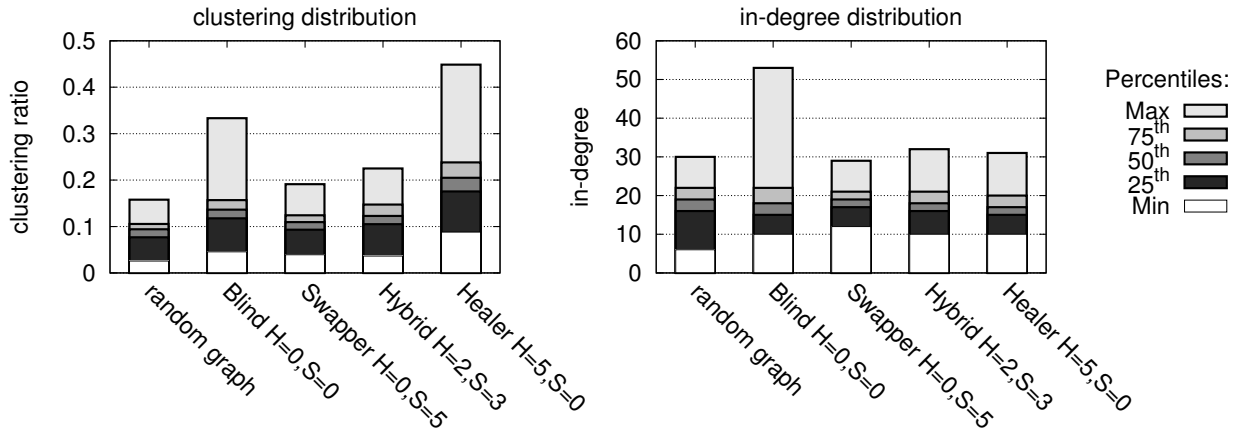


Figure 5.4: PSS: clustering and in-degree for the four strategies and reference random graph.

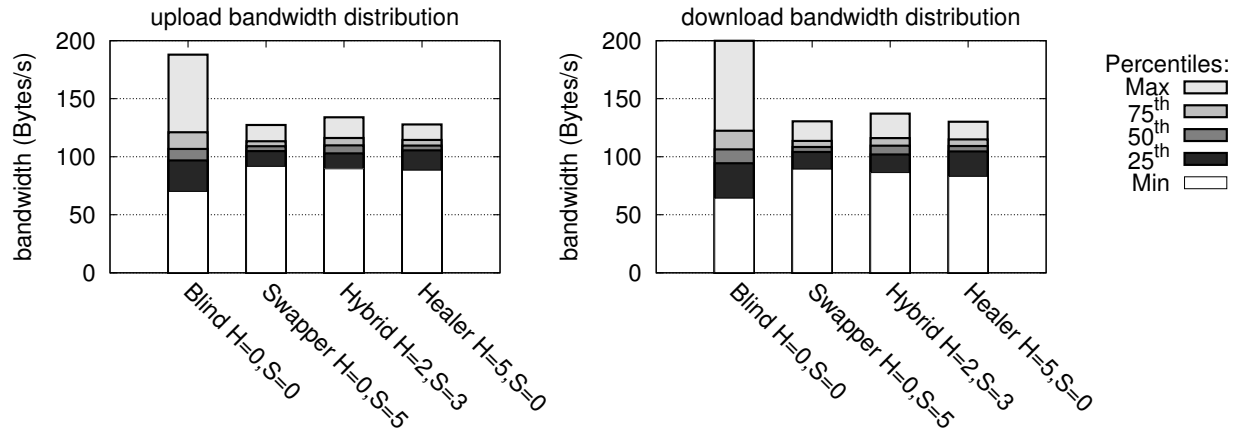


Figure 5.5: PSS: upload and download bandwidth usage distributions.

and load imbalance (directly linked to the in-degree distribution). *Healer* corresponds to Newscast [117], with $H = 0$ and $S = 5$. It favors membership management to randomness. Finally, *Hybrid* mitigates the two objectives by setting $H = 2$ and $S = 3$.

Figure 5.4 presents the quality of randomness of a snapshot of the graph obtained after 5 minutes of execution. The results are presented for each variant as well as for a reference random graph generated offline with the same number of vertices. We present the clustering distribution (left) and in-degree distribution (right). We use a representation based on stacked percentiles throughout this section. The white bar at the bottom represents the minimum value, the pale grey on top the maximal value. Intermediate shades of grey represent the 25^{th}, 50^{th}–the median–, and 75^{th} percentiles. For instance, the median clustering ratio for the *Healer* is 0.2. This means that 50% of the peers have up 20% or less of all of possible pairs of their neighbors that are neighbors themselves.

We observe that, as expected, *Blind* performs badly for both aspects: the clustering is high and the in-degree is skewed, with some peers being in more than 50 views (while the distribution should be as narrow as possible around the average in-degree of 20). The *Swapper* gives the best results in terms of clustering and in-degree, for which it gives similar, respectively, better results than the random graph itself. While the *Healer* performs correctly in terms of in-degree, it yields high clustering, which will result in slower convergence at the

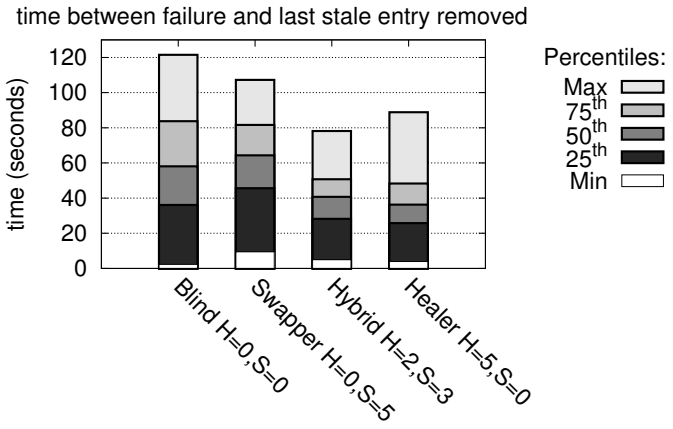


Figure 5.6: PSS: time to remove descriptors of failed peers from the system after half of the network fails.

Topology Construction layer and have a higher chance of leading to partitions. We could not reproduce the skewed in-degree distribution for the Healer presented in [30]. This is due to the smaller size of our deployments compared to what can be obtained in synchronous simulations. The results indicate that the *Swapper* is the best choice in terms of randomness guarantees, while the *Hybrid* seems to be a good compromise with approximate characteristics to that of the random graph.

Next, we evaluate the overhead imposed by the peer sampling layer by observing the consumed upload and download bandwidth. Because the peer sampling service is a low level service it should be as frugal as possible in terms of bandwidth consumption. The active task is invoked every 10 seconds. The distribution of bandwidth usage, both for upload and download, are presented in Figure 5.5. All variants exhibit highly balanced distributions, with the exception of *Blind*. This is clearly a consequence of the high imbalance in in-degree distribution, resulting in some peers being contacted much more or much less than the average.

Finally, we study the quality of the membership management by evaluating the time required for discarding failed peers from the views of all other peers. Until then, the upper layer may try to contact the failed peers for some time, resulting in wasted communication. Note however that the monitoring of peers at the upper layer is performed aggressively through pre-established TCP links, and membership management at the peer sampling layer only has an influence on the efficiency of peer discovery, and on partition resilience (as many stale descriptors in peers' views may lead to a poorly connected graph overall). We use a worst-case scenario where half of the peers (100 out of 200) are removed from the network after a stabilization period of 5 minutes. We observe the views of all peers, reported after each exchange, and in particular the last time each of the failed peers appears in a view. Results in Figure 5.6 indicate that, as expected, *Blind* performs the worst in that respect. Indeed, this strategy does not use the `age` field in the descriptors and thus stale descriptors are discarded only due to random selections after a large number of exchanges. Healer performs the best (in terms of median time), and in particular better than Swapper. This is on par with the conclusions in [30]. This illustrates the compromise made when setting the H and S parameters, which will have an influence on the service given to the upper layers. Again here, *Hybrid* performs well; as mentioned in [30], even a small value of H allows discarding old

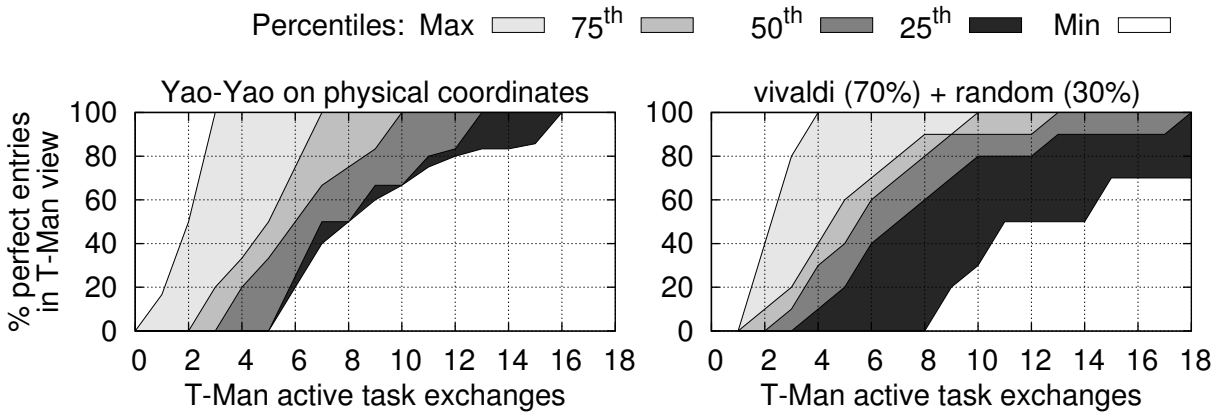


Figure 5.7: TC: Convergence time.

descriptors quickly enough to grant rapid reaction to membership change. As a result, we select the *Hybrid* strategy for the remaining of our evaluation.

5.4.2 Topology Construction Layer

We now evaluate the topology construction layer using the modified version of T-Man [24] we described in Section 5.3.2. We report both the initial metric we used, Vivaldi synthetic coordinates in addition to a few random links, as well as the Yao-Yao graph construction based on geographical distances. In the experiments below the view size is $c = 10$ for the former and $c = k = 8$ cones (view entries) for the latter. The active task period is 10 seconds.

Vivaldi coordinates were generated offline using 5 dimensions (as recommended by [141]) based on the geographical coordinates obtained from PlanetLab. We apply an Euclidean distance function based on these coordinates for $p \times c$ entries of the view, and use random links for the remaining $(1 - p) \times c$ entries. Selecting random links in the view should allow overcoming the inherent clustering and produce connected topologies as per our requirements. For these entries, the pseudo-random distance between any two peers p and q is given by `abs`(`hash`(p)-`hash`(q)). The reason for this is to decouple the distance from any geographical information while making it deterministic (as given by the hash function), allowing each peer to sort every other peer and reach a stable view that can be exposed to the upper layer. It also allows deterministic bidirectional links, another requirement of the upper layer. We were only able to obtain connected topologies when at least $1 - p = 30\%$ of the peers in the view where selected using the pseudo-random criteria. For the Yao-Yao graph, we use the Euclidean distance on the peers geographical positions shown in Figure 5.2. Yao-Yao graphs are always connected.

We present the distribution of convergence time, measured in the number of active task cycles, in Figure 5.7. Convergence measures the time taken for each peer to connect to the ideal peers. These ideal peers are based on an offline pre-computation of the graph. The reported time corresponds to the first inclusion of these peers in the views, and not their exposition to the upper layer, for which we wait 5 more cycles until setting up TCP connections. Interestingly, the Vivaldi/random distance takes longer to converge than the Yao-Yao graph. The reason for this stems from the need to use the pseudo-random criteria. In fact, the topology construction works best when the transitivity among neighbors holds which is not the case when using pseudo-random selection. Unfortunately, this is an inherent

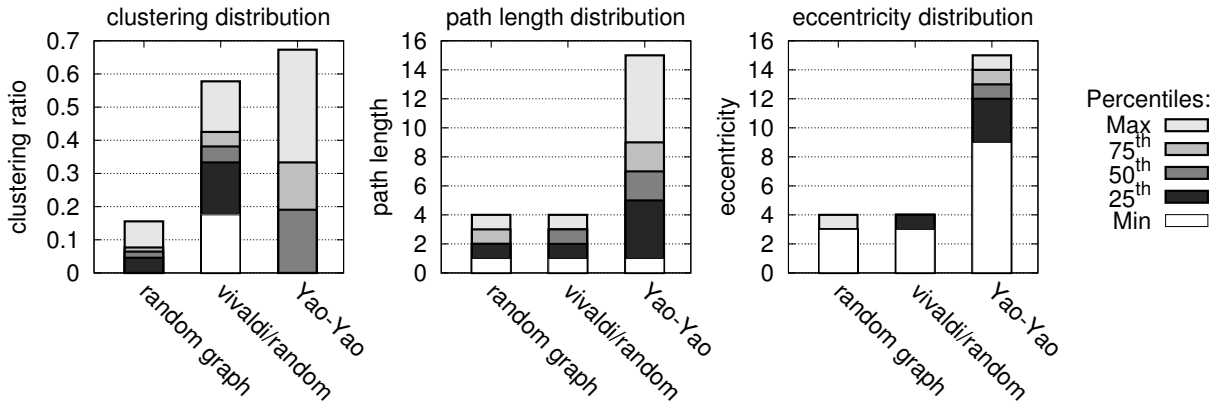


Figure 5.8: TC: characteristics with the two selection metrics and reference random graph.

limitation for metrics based exclusively on the physical distance which can be addressed by the use of planar graphs that take into account the *direction* of neighbors as is the case of the Yao-Yao graph.

Next, we study the graph properties of the topology. We observe in Figure 5.8 the clustering ratio distribution, the path length distribution and the eccentricities distribution. The definition of clustering is the same as above. The path length is the smallest path between any two peers, in number of vertices. The eccentricity of a vertex v in a graph is the maximum distance from v to all other vertices in that graph. A small average path length might result in a fat dissemination tree at the upper layer (with a small maximal depth). However, it does not necessarily corresponds to low end-to-end latencies as the length metric is the number of vertices only. A fat tree also result in a imbalance in the use of peers upload capacities. For instance, the random graph and the Vivaldi/random both present small and even path lengths, but as a result of using long, random links of potentially high delay. The path length distribution for the Yao-Yao graph is higher since only close links are used. More importantly, and as previously noted, guaranteeing a connected network with Vivaldi/random is very difficult and will degenerate, for small systems, in a random graph. Furthermore, the slower convergence means that a link to a failed peer will be replaced slower (in particular for the random portion of the view), putting the system at risk of partition under moderate churn. As a result, we confirm our selection of a Yao-Yao planar-graph as the choice for providing a connected, bidirectional and locality-aware network abstraction to the upper layer.

5.4.3 Dissemination Layer and Client Application

We evaluate the dissemination layer and the client application together. The primary metrics of interest are: the balance of bandwidth usage between the peers, the dissemination delays for individual packets, and the fill ratio at the play buffers at the end peers, which indicates the ability to play the video stream without interruptions or degradations. We let the peer sampling and the topology construction layer stabilize by running the system for 3 minutes without dissemination traffic before sending video packets from a source running VLC as a streaming server, towards all clients, also running VLC as a display client. The first video packet creates the initial dissemination trees, as described in Section 5.3.3. We start our evaluation in a static setting, and then evaluate the behavior of LayStream under churn.

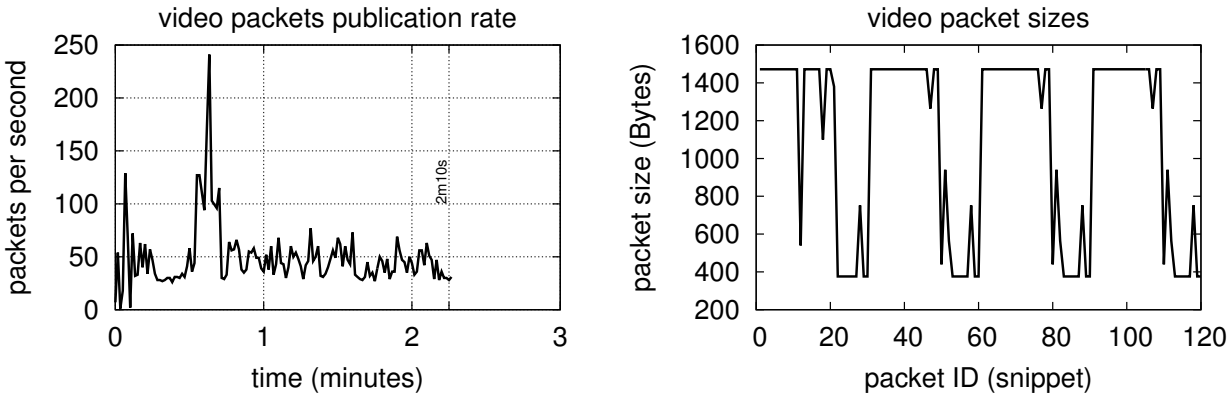


Figure 5.9: Characteristics of the variable bitrate video stream.

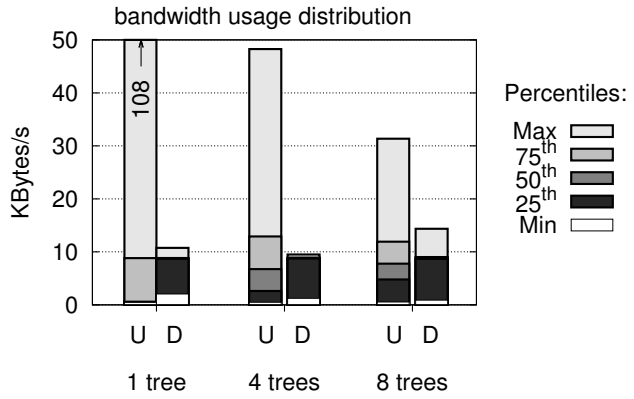


Figure 5.10: Influence of the number of trees on load repartition.

The video streamed by the VLC player at the source peer is a MPEG-encoded video with variable bit rate (VBR). Using VBR allows to use more bits to encode more complex scenes and less bits to encode simpler scenes which generally results in better overall video quality. However, VBR is more challenging to the dissemination layer because it is prone to packet bursts and variable packet sizes. Figure 5.9 presents these characteristics. Around 40 seconds after the start of the video, a succession of fast paced scenes with no visual similarities, result in a burst of messages, that the dissemination trees must handle. We chose to use a relatively short video in order to exhibit the behavior of LayStream upon bootstrap and ending of the stream.

We first evaluate the bandwidth requirements, and in particular the impact of using multiple independent dissemination trees constructed as the source dispatches in a round-robin fashion the packets to 1, 4 or 8 of its Yao-Yao neighbors. Figure 5.10 presents the distribution of the upload (U) and download (D) bandwidth at all the peers, over the complete stream duration while Figure 5.11 presents the upload throughput (as a 5-second moving average over all per-second reports from the 200 peers). Clearly, while the bandwidth requirement is the same in all three cases, the use of a single tree results in a highly skewed upload utilization. This clearly improves when increasing the number of trees. In fact, for 8 trees, half of the peers upload roughly the same amount that they download and the remaining ones only upload slightly more, with a maximal upload requirement of 31 KB/s. Those are typically peers closer to the source (in the topology layer) and as such most of their links

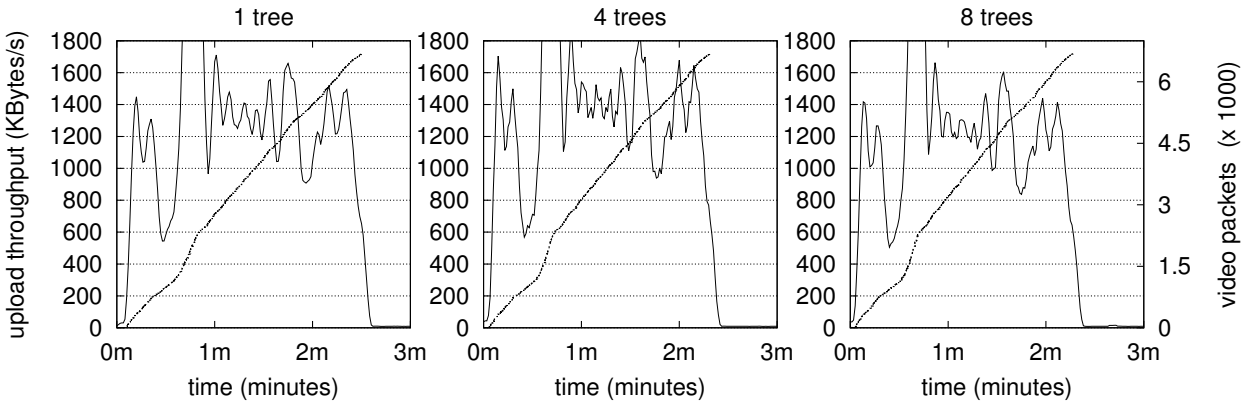


Figure 5.11: Evolution of the upload throughput over time (moving average over five reported measurements)

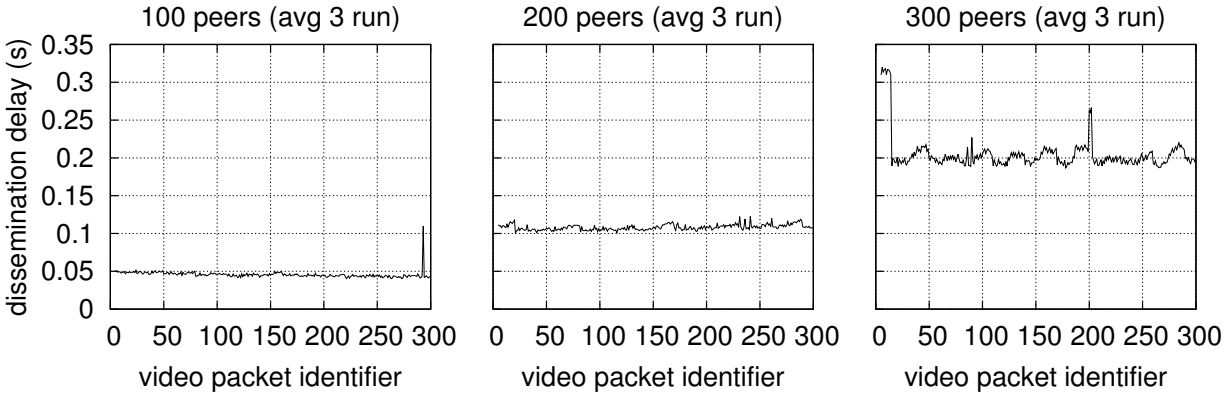


Figure 5.12: Average message dissemination delay for increasing number of peers.

will be active. Still, the improvement from one tree to eight is remarkable when we also take into account that there are no complex load balancing mechanism in place, or explicit construction of disjoint trees [103].

We evaluate the delays for the dissemination of individual packets. Results are presented in Figure 5.12. We present the average dissemination delay for all peers, using a single tree. We vary the number of peers from 100 to 300. Delays for 100 or 200 peers are consistent and around 1 to 2 seconds, which is enough to display the video at end peers with a delay of 3 to 5 seconds as we describe next. Delays are slightly higher and show some spikes in the 300 peers configuration. This is mostly due to the fact that the number of peers is higher than the number of cores, and some packets are subject to scheduling delays near to the top of the tree, resulting in increased delays at all receiving peers.

In the next experiment, we look at the evolution of the buffer occupancy over time. During the experiment, each peer buffers the packets it receives and when required, sends them to VLC. For each packet, we measure the time when it is sent to the VLC client. We first need to determine the minimal number of packets in the buffer for starting the video display, that guarantees a playback without interruption. Through experimental observation we determined the initial buffering requirement to be of 160 packets which is less than 3% of the entire stream (6679 packets). The VLC client starts consuming packets from the buffer when it first contains 160 packets. The buffer occupancy is then a good indicator

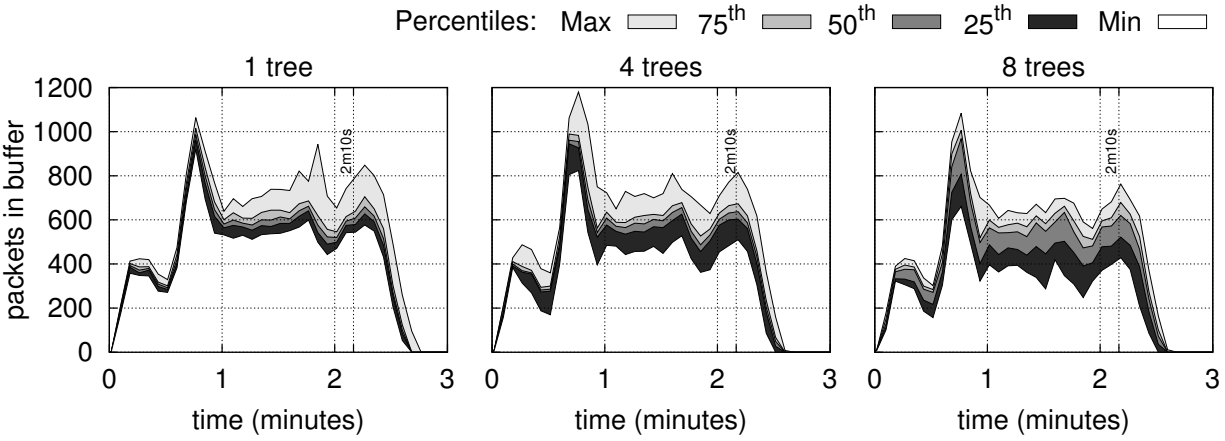


Figure 5.13: Evolution of the distribution of messages in buffers in a static setting.

of LayStream's ability to deliver packets to the video player *on time*. An empty buffer indicates that packets are arriving too slowly and a fault is more likely to result in a pause in the video stream. We present buffer occupancy results for 1, 4 and 8 trees in Figure 5.13. In all scenarios the buffer never gets empty during the dissemination meaning LayStream is able to disseminate the packets to all nodes on time. The larger variance for the 8 trees is due to the increased parallelism in the dissemination that results in more fluctuations on the rate at which packets are received. Still, the buffer occupancy remains at a high enough level to avoid introducing problems in the video quality.

Our last experiment evaluates the ability of the LayStream gossip stack to handle churn. We leverage the churn replay facility and description language of Splay. The average size of the system is still 200 nodes, but we impose that a number corresponding to 1% (moderate) or 2% (heavy) of the peers present in the system leave or join on average, for each 120 seconds period. Figure 5.14.top presents the evolution of the system size for the two cases, and the stacked bars represent joins and leaves to/from the system for each minute. To better assess the impact of churn, we stream a video of 12 minutes, which is a concatenation of 4 instances of the video used for the previous experiments (this can be observed by the four pikes which correspond to the bandwidth-intensive beginning of the video, see Figure 5.9). Figure 5.14.bottom presents the buffer occupancy evolution. As expected, the required buffer size before starting the video playing is larger than in a static setting, but we are able to display the video without interruption, albeit with a larger starting delay. This is due to the fast detection of faults in the topology construction layer, the fast fallback to pre-provisioned backup parents at the dissemination level by sending a reactivation message to the peers permanently connected at the topology construction layer level. The amount of churn has only a moderate impact on the dissemination and buffer occupancy. This is mostly due to the fact that the good convergence property of the topology construction layer and the Yao-Yao graph are able to quickly replace failed entries and sustain the guarantee for the upper dissemination layer in order to the latter to find backup parents upon faults.

We conclude by analyzing the buffer contiguity in both a static and dynamic setting. Buffer contiguity indicates how many packets, starting from the head of the buffer, are consecutive and can thus be promptly delivered to the VLC player. This is complementary with buffer occupancy which only indicates the number of packets and does not consider holes in the packet sequence that might affect the video quality. Figure 5.15 shows the results for 1 and 4 trees for static and moderate churn conditions. LayStream maintains good

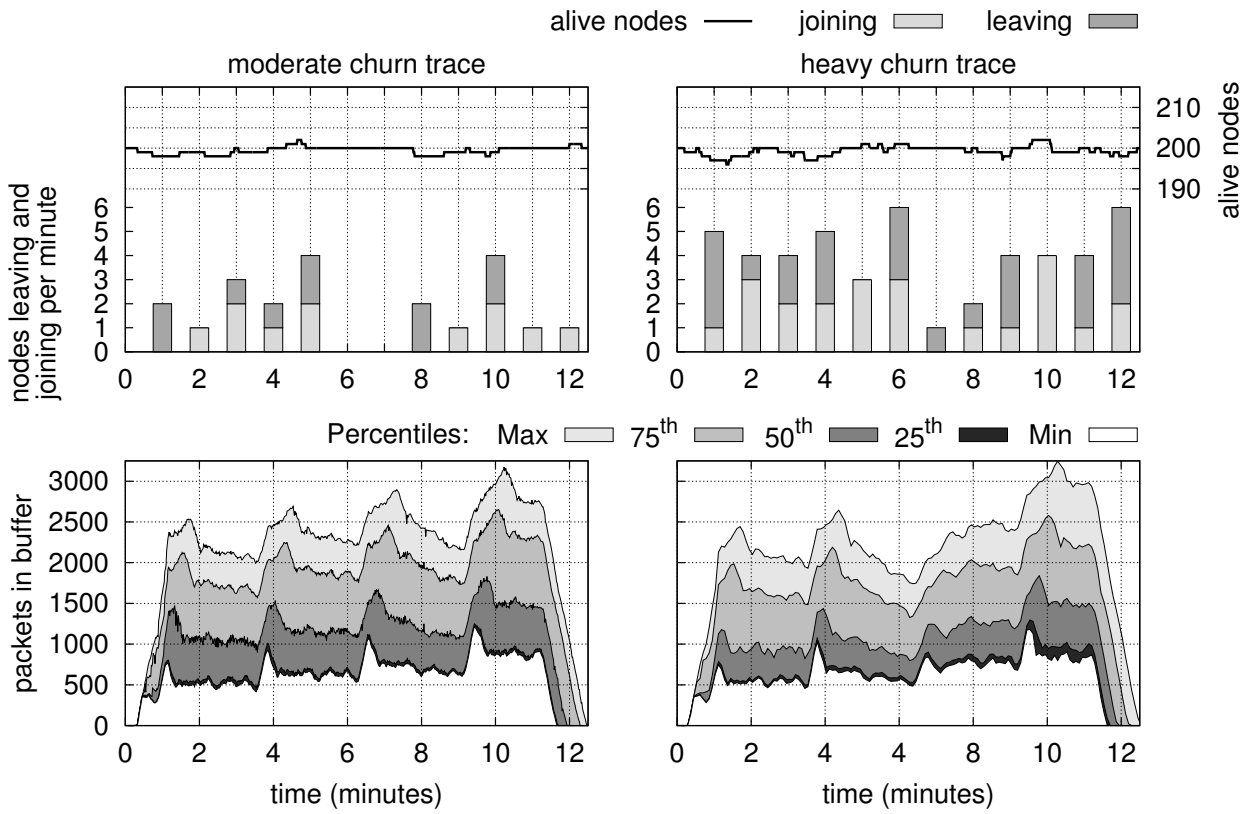


Figure 5.14: Evolution of the distribution of messages in buffers under churn.

buffer contiguity even under churn. This is particularly remarkable when using several trees as, regardless of parallelism, the buffer stores a great portion of contiguous video packets.

5.5 Related Work

There is a large body of research on live video streaming extensively covered in surveys such as [148, 149]. We focus in this section on gossip based approaches to live streaming that were evaluated in non-simulated environments. Most approaches to live streaming, and video dissemination in general, follow a *pull* approach where peers explicitly request data from others. The objective is to preserve bandwidth by avoiding receiving duplicate packets flooding would yield. By placing the burden of retrieving packets on the receivers, pull-based approaches such as GoalBit [150], CoolStreaming [151] and [152] effectively address this problem at the expense of increased latency and communication overhead. GoalBit [150], inspired by BitTorrent, relies on a tracker and requires super peers to do most of the dissemination whereas in LayStream all peers participate equally without any centralized control. This challenge is also addressed in [152], which aims at leveraging the upload bandwidth of the peers while avoiding clogging uplinks. The protocol segments the video in large constant chunks, which is not particularly adapted for live streaming systems that must deal with small packets of non-predictable size packets due to the use of VBR [153].

The alternative to pulling data over unstructured overlays is to push it through trees or forests of trees. Since trees are fragile structures, they must be backed up by an efficient and fast repair mechanism. Approaches such as mTreebone [154] build a tree containing only the peers known to be stable and use an unstructured overlay to cover all peers. In this way,

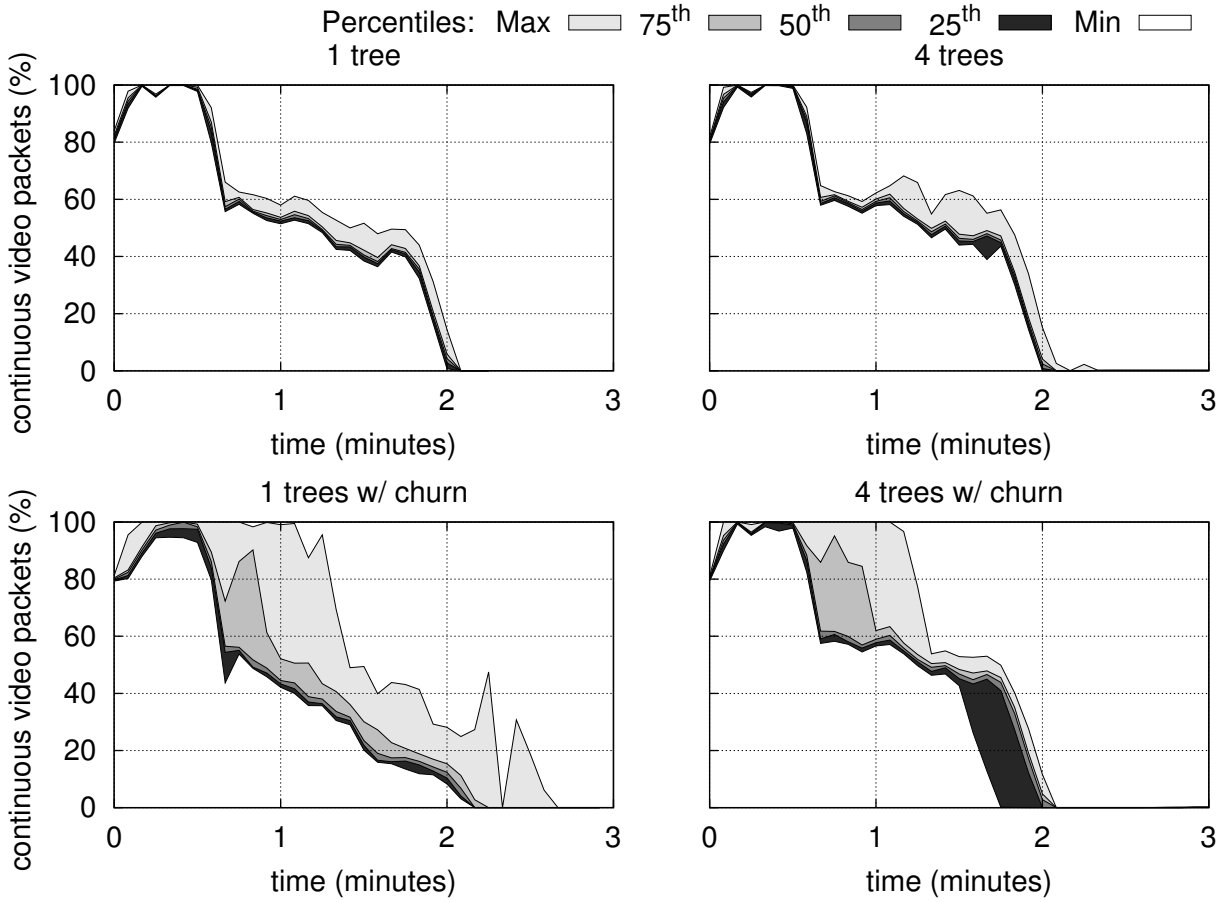


Figure 5.15: Contiguous video segments in buffer.

stable peers can quickly receive the stream by pushing data over the tree while the remaining peers pull data from the overlay as in the protocols above.

Recent work [131, 133] proposed mechanisms to quickly and efficiently construct and maintain trees even under dynamic environments. These tree construction and fast repair principles are used by the LayStream dissemination layer for building forests of trees that encompass all peers. However, these systems assumed an arbitrary random network, without guarantees in terms of network awareness as LayStream does. We have shown in this chapter that the layering of gossip protocols with specific purpose and well-defined guarantees allows us to also meet this requirement.

Finally, the authors of [155] follow a methodology similar to ours in order to assess the possibility to build efficient mesh-based dissemination networks, using live streaming as target application.

5.6 Summary

We studied in this chapter the practicality of building a complex application based on well-established gossip protocols that were previously mostly evaluated by simulation and independently in isolation. We build a system composed of three gossip layers, each implementing a specific abstraction with well-defined guarantees. The complete system offers an autonomous and decentralized reliable service to broadcast VBR video streams from one source to a large number of receivers using the VLC player application. In the process, we

uncovered several issues that stem from the use of simplified assumptions made on simulated environments. Remarkably, even the simple task of building and maintaining a realistic topology fit for video dissemination poses hard challenges and implies important trade-offs. In this spirit we enumerate the problems, lessons learnt, and some solutions encountered when implementing and evaluating each component, in the hope they help other researchers bridge the gap between simulated results and real implementations.

Chapter 6

Conclusion

The design of large-scale distributed systems must consider the awareness to physical and virtual topologies that interconnect the host machines as a key design criteria. Chapters 2 and 3 of this thesis presented protocols and tools to support the design of distributed systems by incorporating the network topology constraints in the system design itself. Chapters 4 and 5 presented large-scale data-dissemination applications that carefully organize the virtual topology of the nodes to minimize the redundant network traffic or the message reception' delays at each node. The obtained virtual topology still provides the properties of connectivity and fault-tolerance required by the target large-scale networks.

This chapter summarizes the contributions presented in this dissertation. Finally, this thesis concludes by presenting some of the open research perspectives and possible future work.

6.1 Summary of contributions

The contributions of this thesis can be summarized as follows.

We described Whisper, a fully decentralized approach to confidential group communication in large-scale systems in which the traffic may have to take multi-hops paths to circumvent network topology limitations, such as NAT devices and firewalls. Whisper supports the creation of confidential communication routes without the need for a trusted third party. Moreover, it provides membership management and overlay maintenance among private groups of nodes communicating in a confidential manner. Our prototype implementation is evaluated in real-world settings. The results indicate that the price of confidentiality remains reasonable in terms of network load and processing costs.

We proposed SplayNet, a novel approach to topology emulation with user-space mechanisms and without specific support from the underlying operating system. The approach features configurable and modular network models, support for complex topologies with inner routers, link sharing models and overhead emulation. A fully decentralized monitoring algorithm allows SplayNet to emulate congestion for inner links in the topology without instantiating inner nodes nor requiring a centralized control point, as other state-of-the-art systems do. We evaluate our implementation on micro-benchmarks as well as complex distributed protocols and applications. The results indicate that SplayNet achieves high accuracy for network emulation with low resource requirements, support for concurrent deployments and multi-users, high-scalability and minimum management efforts.

We presented Brisa, an efficient, robust and scalable data dissemination system. Brisa leverages the robustness and scalability of an epidemic substrate to build efficient dissemination structures that are correct, i.e., cover all nodes, by construction. Such structures

are built in a distributed fashion with local knowledge only and with minimal overhead. BRISA has been designed in a way that upon failures or churn in the underlying network topology, the data dissemination structures are quickly repaired leveraging the underlying epidemic substrate. The system supports several data dissemination structures: trees, directed acyclic graphs (DAGs) and forests of trees. The evaluation of BRISA was carried on PlanetLab and on a local cluster comparing it with state-of-the-art data dissemination systems from the literature. The results indicate that BRISA is a robust and efficient system for data-intensive applications designed for large-scale networks.

We introduced LAYSTREAM, a complex application built from the ground-up by stacking non-trivial gossip-based protocols. The use-case was a live video streaming, with strict requirements on performance and robustness. LAYSTREAM relies on a PSS layer, a topology construction service and a dissemination service. It re-uses some of the mechanisms proposed by BRISA to efficiently disseminate video stream packets on a large-scale network. Our evaluation takes into account real-world settings. This study uncovered several issues related to the stacking of gossip-based protocols, usually overlooked by literature due to the simplified nature of their simulation-based evaluation.

6.2 Perspectives

This concluding section presents some of the research perspectives opened by the work presented in this thesis. We organize them according to their specific domains of applicability.

6.2.1 Confidentiality and anonymity

Confidentiality and anonymity for data streaming systems on large-scale networks. Our research on WHISPER presents protocols and distributed systems targeting applications such as group membership and censorship-resistant chats with relatively modest bandwidth requirements, especially for today's Internet standards. Conversely, data streaming applications, as those motivating our research on LAYSTREAM-like systems, require high bandwidth throughput to sustain superior perceived quality of service (QoS) by end-users. It is an open research problem to apply the techniques used in WHISPER to more demanding scenarios, such as live video streaming applications similar to LAYSTREAM. The typical scenario to target by the resulting system will be a large-scale deployment of a fully-decentralized video streaming system for sensible-content or pay-per-view channels, with stringent confidentiality requirements. Such protocols and techniques will offer confidentiality and anonymity guarantees as well as adequate QoS guarantees to high-throughput streaming systems targeting large-scale networks. A possible way to tackle this problem is to take inspiration from scalable coding and progressive encryption techniques [156].

Privacy-preserving group communications under stronger adversarial models. WHISPER explored the creation of anonymous and privacy-preserving communication channels under a *curious but honest* adversary model. Nevertheless, higher levels can be considered for the adversaries. For instance, it would be possible for an attacker to deploy a *forwarding attack*, where nodes can simply stop forwarding messages. Although the detection of such attacks was studied in the context of wireless networks [157–159], it remains an open problem to protect systems like WHISPER from these kind of adversaries and attacks. It will be interesting to study the applicability of the solutions proposed for wireless networks in the scenarios addressed by WHISPER.

Alternatives to Chaum's mixes. WHISPER exploits Chaum's mixes as a technique to build onion-routing paths. This technique is powerful in its simplicity and makes it an ideal candidate for implementors. One of the main drawbacks is that the paths are chosen in advance by the sender of the messages. One possible way to overcome this limitation is to rely on a *crowd system* [160], where the responsibility to build the routing paths is delegated to the routers. This approach could open interesting perspectives in terms of *probabilistic anonymity* guarantees.

6.2.2 Data dissemination

Capacity-aware topologies for resource-efficient data dissemination. The chapters on BRISA and LAYSTREAM presented techniques to build efficient dissemination topologies, such as trees or DAGs. These topologies define the routing paths that packets will follow to reach every node taking part in the dissemination of the data. We explored several strategies to build such topologies, presented in the context of BRISA and LAYSTREAM, respectively in Section 4.2.5 and Section 5.3.3. One of the areas left unexplored by our work is define and evaluate strategies that take into account the network capabilities of each node. The intuition behind this idea is that nodes with high-capacity links should contribute more than others to the dissemination of data to other nodes. Existing systems [161] exploit a similar idea to boost the download time of files by forming groups of nodes that collaborate to download a single file. We are interested in adapting similar techniques to the context of live data streaming.

6.2.3 User-space emulation

The approach to user-space topology emulation exploited by SPLAYNET proved to be successful and efficient. It paves the road to apply the same approach to other domains, each with its own challenges and open issues. We suggest two of such possible extensions.

User-space wireless, mobile and dynamic topology emulation. The most direct perspective to extend the SPLAYNET tools and mechanisms is the integration of wireless and mobile emulation capabilities, as well as the support for non-fully connected and dynamic topologies. SPLAYNET will then allow experimenters to emulate protocols for sensor and ad-hoc networks, for which building a physical testbed is a complex and costly operation. To tackle this problem, the topology description language must be extended to define the range of the wireless antenna, as well the trajectory of the node. Then, each node will receive, as part of a regular job submission, all the known trajectories and can easily compute a *reachability timeline*, i.e. to know at any moment which other nodes in the network are reachable via message emission.

User-space modular energy emulation framework. A great deal of research investments are being devoted toward the so-called *green computing*. With this perspective in mind, we envision a modular energy model to plug to each node to easily evaluate the energy impact of distributed applications. Each node can be paired with a different energy module, specified in a similar manner as the network features or the wireless characteristics. Researchers will be able to tune the performances and analyzing the trade-offs of different configuration factors with respect to the application's energy requirements. One of the challenges opened by this perspective is the correct definition of energy models. One possible solution is to base the definition of such models upon measurement studies [162].

6.2.4 Network topology discovery

Crowd-sourced generation and validation of topology models. One of the major difficulties in testing distributed systems under realistic network topology conditions is the lack of realistic topologies extracted from Internet's routers. BGP route dumps[1] are publicly available, but it is not straightforward to integrate them into an easy-to-use tool to study the behavior of distributed systems under a realistic internet routing model. We envision the possibility to gather measures of network topologies under a crowd-sourced model. These measures can be used as input to topology generators to build realistic topology models. The resulting topology models, and the associated model instances, will however prove to be challenging for the current network emulators, in particular due to the expected scale, orders of magnitude bigger than the currently available ones. We embrace such challenging scenarios: they represent an opportunity to improve and optimize the emulation techniques contributed by SplayNet.

[1] `https://labs.ripe.net`

Appendix A

Publications

The publications authored in the context of this thesis are highlighted by a ⋆ symbol before the respective title.

Valerio Schiavoni, Etienne Rivière, Pascal Felber.
⋆ **SplayNet: Distributed User-Space Topology Emulation.**
In the *Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference* (Middleware 2013). Beijing, China, December 2013.

Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, Etienne Rivière, Valerio Schiavoni and Josè Valerio.
CoFeed: privacy-preserving Web search recommendation based on collaborative aggregation of interest feedback.
In *Software: Practice and Experience* (SP&E 2011). Volume 43, Issue 10, October 2013.

Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, Etienne Rivière.
⋆ **Lightweight, Efficient, Robust Epidemic Dissemination.**
In *Journal of Parallel and Distributed Computing* (JPDC). Volume 7, Issue 7, July 2013.

Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni and Jean-Bernard Stefani.
A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. In *Software: Practice and Experience* (SP&E 2011). Volume 42, Issue 5, May 2012.

Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, Etienne Rivière.
⋆ **BRISA: Combining Efficiency and Reliability in Epidemic Data Dissemination.**
In the *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium* (IPDPS 2012). Shangai, China, May 2012. **Best Paper Award**.

Alessio Pace, Vivien Quèma, and Valerio Schiavoni.
Exploiting Node Connection Regularity for DHT Replication.
In the *Proceedings 30th IEEE International Symposium on Reliable Distributed Systems* (SRDS 2011). Madrid, Spain, October 2011.

Valerio Schiavoni, Etienne Rivière, Pascal Felber.
⋆ **Whisper: Middleware for Confidential Communication in Large-Scale Networks.**
In the *Proceedings of the 31th IEEE International Conference on Distributed Computing Systems* (ICDCS 2011). Minneapolis, Minnesota, USA, June 2011.

Valerio Schiavoni, Etienne Rivière.
⋆ **Whisper: communications confidentielles dans un monde décentralisé.**
In the *Proceedings of the 13es Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications)* (ALGOTEL 2011). Cap Estérel, France, May 2011.

Pascal Felber, Martin Rajman, Etienne Rivière, Valerio Schiavoni, José Valerio.
SPADS: Publisher Anonymization for DHT Storage.
In the *Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing* (P2P 2010). Delft, The Netherlands, August 2010.

Anne-Marie Kermarrec, Alessio Pace, Vivien Quéma, Valerio Schiavoni.
NAT-resilient Gossip Peer Sampling.
In the *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems* (ICDCS 2009). Montreal, Canada, June 2009.

Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, Jean-Bernard Stefani.
Reconfigurable SCA Applications with the FraSCAti Platform.
In the *Proceedings of 6th IEEE International Conference on Services Computing* (SCC 2009). Bangalore, India, September 2009.

Valerio Schiavoni and Vivien Quéma.
A posteriori defensive programming: an annotation toolkit for DoS-resistant component-based architectures.
In the *Proceedings the 21st Annual ACM Symposium on Applied Computing* (SAC 2006). Dijon, France, April, 2006.

A.1 Posters

Valerio Schiavoni, Etienne Rivière, Pascal Felber.
⋆ **User-Space Network Emulation for the SPLAY Platform.**
In the *Proceedings of the **poster** session of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2012). London, UK, March 2012.

Valerio Schiavoni, Etienne Rivière, Pascal Felber.
⋆ **The private peer sampling service: the ground for your secret society.**
In the *Proceedings of the **poster** session of 9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI 2010). Vancouver, Canada, October 2010.

References

[1] I. Global, "Traffic Forecast and Methodology, 2006-2011", *Cisco white paper*, 2008.

[2] A. Odlyzko, "Internet pricing and the history of communications", *Computer Networks*, 36(5):493–517, Elsevier, 2001.

[3] J. Nielsen, "Nielsens Law of Internet Bandwidth. Alertbox: Jakob Nielsens Column on Web Usability", 1998.

[4] M. Eagles and R. Cruickshank, "The Cost of Fair Bandwidth: Business Case on the Rise of Online Video, P2P, and Over-the-Top; Proactive Steps Operators can Take Today to Meet Bandwidth Demands in the New Future", *NCTA Technical Papers*, 2007.

[5] "Bureau of Labor Statistics", `http://www.bls.gov`, Last accessed: February, 2014.

[6] M. Ripeanu, "Peer-to-Peer architecture case study: Gnutella network", *P2P'01: Proceedings of the First International Conference on Peer-to-Peer Computing*, pp. 99–100, IEEE, 2001.

[7] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload", *ACM SIGOPS Operating Systems Review*, volume 37, pp. 314–329, ACM, 2003.

[8] D. Qiu and R. Srikant, "Modeling and performance analysis of BitTorrent-like peer-to-peer networks", *ACM SIGCOMM Computer Communication Review*, 34(4):367–378, ACM, 2004.

[9] Y. Kulbak, D. Bickson, et al., "The eMule protocol specification", http://www.sourceforge.net/projects/emule.

[10] D. Bonfiglio, M. Mellia, M. Meo, N. Ritacca, and D. Rossi, "Tracking down Skype traffic", *INFOCOM'08: Proceedings of the 27th Conference on Computer Communications*, pp. 261–265, IEEE, 2008.

[11] T. Silverston and O. Fourmaux, "P2P IPTV measurement: a case study of TVants", *CoNEXT'06: Proceedings of the Conference on Emerging Networking EXperiments and Technologies*, p. 45, ACM, 2006.

[12] G. Huang, "PPLive: A practical P2P live system with huge amount of users", *Proceedings of the ACM SIGCOMM Workshop on Peer-to-Peer Streaming and IPTV Workshop*, 2007.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications", *IEEE/ACM Transaction on Networking*, 11(1):17–32, Feb 2003.

[14] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers", *ACM SIGCOMM Computer Communication Review*, 40(4):51–62, ACM, 2010.

[15] L. Leonini, E. Rivière, and P. Felber, "SPLAY: Distributed Systems Evaluation Made Simple (or how to turn ideas into live systems in a breeze)", *NSDI'09: Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pp. 185–198, USENIX, April 2009.

[16] B. Cohen, "Incentives build robustness in BitTorrent", *Workshop on Economics of Peer-to-Peer systems*, volume 6, pp. 68–72, 2003.

[17] P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast", *ACM Transactions on Computer Systems*, 21:341–374, ACM, 2003.

[18] W. Vogels, R. van Renesse, and K. Birman, "The power of epidemics: robust communication for large-scale distributed systems", *SIGCOMM Computer Communication Review*, 33(1):131–135, ACM Press, New York, NY, USA, 2003.

[19] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip", *OSDI'06: Proceedings of the 9th International Symposium on Operating Systems Design and Implementation*, pp. 191–204, USENIX, November 2006.

[20] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks", *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[21] S. Kashyap, S. Deb, K. V. M. Naidu, R. Rastogi, and A. Srinivasan, "Efficient gossip-based aggregate computation", *PODS '06: Proceedings of the 25th Symposium on Principles of Database Systems*, pp. 308–317, New York, NY, USA, 2006, ACM.

[22] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-Based Computation of Aggregate Information", *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, p. 482, Washington, DC, USA, 2003, IEEE Computer Society.

[23] D. Kostoulas, D. Psaltoulis, I. Gupta, K. P. Birman, and A. J. Demers, "Active and passive techniques for group size estimation in large-scale and dynamic distributed systems", *Journal of Systems and Software*, 80(10):1639–1658, Elsevier Science Inc., New York, NY, USA, 2007.

[24] M. Jelasity, A. Montresor, and O. Babaoglu, "T-Man: Gossip-based fast overlay topology construction", *Computer Networks*, 53(13):2321–2339, August 2009.

[25] R. Guerraoui, S. B. Handurukande, K. Huguenin, A.-M. Kermarrec, F. L. Fessant, and E. Rivière, "GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles", *P2P'06: Proceedings of the 6th International Conference on Peer-to-Peer Computing*, IEEE, 2006.

[26] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse, "Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead", *IPTPS'03: Proceedings of 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[27] A. Montresor, M. Jelasity, and O. Babaoglu, "Chord on Demand", *P2P'05: Proceedings of the 5th International Conference on Peer-to-Peer Computing*, IEEE, 2005.

[28] R. van Renesse, Y. Minsky, and M. Hayden, "A Gossip-style Failure Detection Service", *Middleware'98: Proceedings of the IFIP Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 55–70, Springer, 1998.

[29] A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal, "Distributed slicing in dynamic systems", *ICDCS'07: Proceedings of the 27th International Conference on Distributed Computing Systems*, IEEE, 2007.

[30] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling", *ACM Transactions on Computer Systems*, 25(3), August 2007.

[31] S. Voulgaris, D. Gavidia, and M. V. Steen, "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays", *Journal of Network and Systems Management*, 13(2):197–217, Kluwer Academic Publishers, June 2005.

[32] V. Schiavoni, E. Rivière, and P. Felber, "WHISPER: Middleware for Confidential Communication in Large-Scale Networks", *ICDCS'11: Proceedings of the 31st International Conference on Distributed Computing Systems*, pp. 456–466, IEEE, 2011.

[33] V. Schiavoni, E. Riviere, and P. Felber, "SplayNet: Distributed User-Space Topology Emulation", *Middleware'13: Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Springer, 2013.

[34] M. Matos, V. Schiavoni, P. Felber, R. Oliveira, and E. Riviere, "BRISA: Combining Efficiency and Reliability in Epidemic Data Dissemination", *IPDPS'12: Proceedings of the 26th International Parallel and Distributed Processing Symposium*, pp. 983–994, IEEE, 2012.

[35] M. Matos, V. Schiavoni, P. Felber, R. Oliveira, and E. Rivière, "Lightweight, efficient, robust epidemic dissemination", *Journal of Parallel and Distributed Computing*, 73(7):987–999, Elsevier, 2013.

[36] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router", *Security'04: 13th USENIX Security Symposium*, pp. 303–320, San Diego, CA, USA, 2004.

[37] D. M. Goldschlag, M. G. Reed, and P. F. Syverson, "Hiding Routing Information", *First International Workshop on Information Hiding*, pp. 137–150, 1996.

[38] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms", *Commun. ACM*, 24(2):84–90, ACM, New York, NY, USA, 1981.

[39] M. Casado and M. J. Freedman, "Peering through the Shroud: The Effect of Edge Opacity on IP-based Client Identification", *NSDI'07: Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, USENIX, April 2007.

[40] R. Roverso, S. El-Ansary, and S. Haridi, "NATCracker: NAT Combinations Matter", *ICCN'09: Proceedings of the 18th International Conference on Computer Communications and Networks*, volume 0, 2009.

[41] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators", *ATC'05: Proceedings of the Annual Technical Conference*, USENIX, 2005.

[42] A.-M. Kermarrec, A. Pace, V. Quéma, and V. Schiavoni, "NAT-resilient Gossip Peer Sampling", *ICDCS'2009: Proceedings of the 29th International Conference on Distributed Computing Systems*, pp. 360–367, IEEE, June 2009.

[43] CISCO IP NAT translation timeouts specification: `http://www.cisco.com/en/US/docs/ios/ipaddr/command/reference/iad_nat.html#wp1076124`.

[44] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron, "Cashmere: resilient anonymous routing", *NSDI'05: Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pp. 301–314, USENIX, 2005.

[45] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach, "AP3: cooperative, decentralized anonymous communication", *Proceedings of the 11th ACM SIGOPS European workshop*, 2004.

[46] M. Kondo, S. Saito, K. Ishiguro, H. Tanaka, and H. Matsuo, "Bifrost: A Novel Anonymous Communication System with DHT", *Proceedings of PDCAT '09*, pp. 324–329, 2009.

[47] P. Felber, M. Rajman, E. Rivière, V. Schiavoni, and J. Valerio, "SPADS: Publisher Anonymization for DHT Storage", *P2P'10: 10th International Conference on Peer-to-Peer Computing*, IEEE, 2010.

[48] M. Freedman, E. Sit, J. Cates, and R. Morris, "Introducing tarzan, a peer-to-peer anonymizing network layer", *Peer-to-Peer Systems*, Springer, 2002.

[49] S. Katti, J. Cohen, and D. Katabi, "Information Slicing: Anonymity Using Unreliable Overlays", *NSDI'07: Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, USENIX, 2007.

[50] S. Goel, M. Robson, M. Polte, and E. G. Sirer, "Herbivore: A Scalable and Efficient Protocol for Anonymous Communication", TR2003-1890, Cornell University Comp. and Inf. Science, feb 2003.

[51] R. Sherwood, B. Bhattacharjee, and A. Srinivasan, "P5: a protocol for scalable anonymous communication", *J. Comput. Secur.*, 13:839–876, Amsterdam, The Netherlands, The Netherlands, December 2005.

[52] K. P. Kihlstrom and R. S. Elliott, "Performance of an Intrusion-Tolerant Gossip Protocol", *PDCS'09: 21st International Conference on Parallel and Distributed Computing and Systems*, IASTED, 2009.

[53] G. P. Jesi, A. Montresor, and M. van Steen, "Secure Peer Sampling", *Elsevier Computer Networks - Special Issue on Collaborative Peer-to-Peer Systems*, 54(12):2086–2098, 2010.

[54] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine resilient random membership sampling", *Computer Networks*, 53(13):2340–2359, aug 2009.

[55] H. Johansen, A. Allavena, and R. van Renesse, "Fireflies: scalable support for intrusion-tolerant network overlays", *EuroSys'06: 1st ACM SIGOPS European Conference on Computer Systems*, 2006.

[56] A. Bakker and M. van Steen, "PuppetCast: A Secure Peer Sampling Protocol", *Proc. of the European Conference on Computer Network Defense (EC2ND 2008)*, pp. 3–10, Dublin, Ireland, dec 2008.

[57] J. Leitão, R. van Renesse, and L. Rodrigues, "Balancing gossip exchanges in networks with firewalls", *IPTPS'10: 9th International Workshop on Peer-to-Peer Systems*, 2010.

[58] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy, "The Gossple Anonymous Social Network", *Middleware'10: 11th International Middleware Conference*, Bangalore, India, December 2010, ACM/IFIP/USENIX.

[59] O. Heen, E. L. Merrer, C. Neumann, and S. Onno, "Distributed and Private Group Management", *SRDS'12: Proceedings of the 31st Symposium on Reliable Distributed Systems*, pp. 191–200, IEEE, 2012.

[60] D. Dolev and A. Yao, "On the Security of Public Key Protocols", *IEEE Transactions on Information Theory*, 29(2):198–208, IEEE Press, Piscataway, NJ, USA, September 2006.

[61] A. Singh, G. Urdaneta, M. van Steen, and R. Vitenberg, "On Leveraging Social Relationships for Decentralized Privacy-preserving Group Communication", *Proceedings of the Fifth Workshop on Social Network Systems*, SNS '12, pp. 5:1–5:6, New York, NY, USA, 2012, ACM.

[62] "PlanetLab", `http://www.planet-lab.org`, Last accessed: February, 2014.

[63] E. Biersack, P. Rodriguez, and P. Felber, "Performance analysis of peer-to-peer networks for file distribution", *QofIS'04: 5th Workshop on Quality of Future Internet Services*, 2004.

[64] C. Gkantsidis and P. Rodriguez, "Network coding for large scale content distribution", *INFOCOM'05: Proceedings of the 24th Annual Joint Conference of the Computer and Communications Societies*, IEEE, 2005.

[65] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: exploiting in-network aggregation for big data applications", *NSDI'12: Proceedings of the 9th Symposium on Networked Systems Design and Implementation*, pp. 3–3, USENIX, 2012.

[66] S. Kristiansen, T. Plagemann, and V. Goebel, "Towards scalable and realistic node models for network simulators", *ACM SIGCOMM Computer Communication Review*, pp. 418–419, 2011.

[67] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and Accuracy in a Large-scale Network Emulator", *OSDI'02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pp. 271–284, ACM, 2002.

[68] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks", *OSDI'02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pp. 255–270, ACM, 2002.

[69] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale virtualization in the Emulab network testbed", *ATC'08: Proceedings of the Annual Technical Conference*, USENIX, 2008.

[70] E. Weingärtner, F. Schmidt, H. Lehn, T. Heer, and K. Wehrle, "SliceTime: a platform for scalable and accurate network emulation", *NSDI'11: Proceedings of the 8th Symposium on Networked Systems Design and Implementation*, USENIX, 2011.

[71] M. Carson and D. Santay, "NIST Net-A Linux-based network emulation tool", *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[72] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau, "The Flexlab Approach to Realistic Evaluation of Networked Systems", *NSDI'07: Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, USENIX, 2007.

[73] L. Nussbaum and O. Richard, "Lightweight emulation to study peer-to-peer systems", *Concurrency and Computation: Practice and Experience*, 20(6):735–749, 2008.

[74] Z. Puljiz, R. Penco, and M. Mikuc, "Performance analysis of a decentralized network simulator based on IMUNES", *SPECTS'08: International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2008.

[75] M. Pizzonia and M. Rimondini, "Netkit: easy emulation of complex networks on inexpensive hardware", *TridentCom'08: 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2008.

[76] "NetEM", `www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[77] D. Ingham and G. Parrington, "Delayline: a wide-area network emulation tool", *Computing Systems*, 7(3):313–332, 1994.

[78] R. Roverso, M. Al-Aggan, A. Naiem, A. Dahlstrom, S. El-Ansary, M. El-Beltagy, and S. Haridi, "MyP2PWorld: Highly Reproducible Application-Level Emulation of P2P Systems", *SASOW'08: 2nd International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, IEEE, 2008.

[79] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo, "WiDS: an integrated toolkit for distributed system development", *HotOS'10: 10th Workshop on Hot Topics in Operating Systems*, USENIX, 2005.

[80] M. Avvenuti and A. Vecchio, "Application-level network emulation: the EmuSocket toolkit", *Journal of Network and Computer Applications*, 29(4):343–360, 2006.

[81] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks", *HotNets'10: 9th Workshop on Hot Topics in Networks*, ACM, 2010.

[82] Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe, and S. Sekiguchi, "GNET-1: Gigabit ethernet network testbed", *ClusterComp'04: Proceedings of the 6th International Conference on Cluster Computing*, IEEE, 2004.

[83] "GEM: network impairment emulator", `http://www.spirent.com/Networks-and-Applications/Network_Emulation_Impairments`.

[84] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen, "Seawind: a wireless network emulator", *MMB'01: Proceedings of the 11th GI/ITG Conference on Measuring, Modeling and Evaluation of Computer and Communication Systems*, 2001.

[85] P. Zheng and L. M. Ni, "Empower: A network emulator for wireline and wireless networks", *INFOCOM'03: 22dn Annual Joint Conference of the Computer and Communications Societies*, volume 3, pp. 1933–1942, IEEE, 2003.

[86] H. Tazaki and A. H., "DNEmu: Design and Implementation of Distributed Network Emulation for Smooth Experimentation", *TridentCom'12: Proceedings of the 8th ICST International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2012.

[87] M. Carbone and L. Rizzo, "Dummynet revisited", *ACM SIGCOMM Computer Communication Review.*, 40(2):12–20, 2010.

[88] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization", *SOSP'03: Proceedings of the 19th Symposium on Operating Systems Principles*, pp. 164–177, ACM, 2003.

[89] M. Eriksen, "Trickle: A userland bandwidth shaper for unix-like systems", *ATC'05: Proceedings of the Annual Technical Conference*, USENIX, 2005.

[90] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. Snoeren, "Cloud control with distributed rate limiting", *ACM SIGCOMM Computer Communication Review.*, volume 37, pp. 337–348, ACM, 2007.

[91] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork", *INFOCOM'96: Proceedings of the 15th Annual Joint Conference of the Computer and Communications Societies*, IEEE, 1996.

[92] K. Fall, "Network emulation in the Vint/NS simulator", *ISCC'99: Proceedings of the 4th International Symposium on Computers and Communications*, IEEE, 1999.

[93] R. Ricci, C. Alfeld, and J. Lepreau, "A solver for the network testbed mapping problem", *ACM SIGCOMM Computer Communication Review*, 33(2):65–81, 2003.

[94] E. Coffman Jr, M. Garey, and D. Johnson, "Approximation algorithms for bin packing: A survey", *Approximation algorithms for NP-hard problems*, PWS Publishing, 1996.

[95] Q. Yin and T. Roscoe, "VF2x: Fast, efficient virtual network mapping for real testbed workloads", *TridentCom'12: Proceedings of the 8th ICST International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2012.

[96] Y. Etsion, D. Tsafrir, and D. Feitelson, "Effects of clock resolution on the scheduling of interactive and soft real-time processes", *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pp. 172–183, ACM, 2003.

[97] P. P. Tang and T.-Y. Tai, "Network traffic characterization using token bucket model", *INFOCOM'99: Proceedings of the 18th Annual Joint Conference of the Computer and Communications Societies.*, volume 1, pp. 51–62, IEEE, 1999.

[98] D. Bertsekas and R. Gallager, *Data Networks*, Prentice-Hall, 1992.

[99] D. M. Chiu, "Some Observations on Fairness of Bandwidth Sharing", *ISCC'00: 5th Symposium on Computers and Communications*, IEEE, 2000.

[100] F. P. Kelly, "Charging and rate control for elastic traffic", *European Transactions on Telecommunications*, 8:33–37, 1997.

[101] L. Massoulié and J. Roberts, "Bandwidth sharing: objectives and algorithms", *IEEE/ACM Transactions on Networking*, 10(3):320–328, IEEE, 2002.

[102] M. Heusse, S. A. Merritt, T. X. Brown, and A. Duda, "Two-way TCP connections: old problem, new insight", *SIGCOMM Computer Communication Review*, 41(2):5–15, April 2011.

[103] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: high-bandwidth multicast in cooperative environments", *SOSP'03: Proceedings of the 19th Symposium on Operating Systems Principles*, pp. 298–313, ACM, 2003.

[104] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks", *Science*, 286:509–512, 1999.

[105] J. F. Gantz and C. Chute, "The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011", *IDC*, 2008.

[106] D. Frey, R. Guerraoui, A.-M. Kermarrec, M. Monod, and V. Quéma, "Stretching gossip with live streaming", *DSN'09: Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks*, IEEE, 2009.

[107] "Murder: Fast datacenter code deploys using BitTorrent", `https://blog.twitter.com/2010/murder-fast-datacenter-code-deploys-using-bittorrent`, Last accessed: February, 2014.

[108] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure", *IEEE Journal on Selected Areas in Communications*, 20:1489–1499, IEEE, 2002.

[109] J. Liang, S. Ko, I. Gupta, and K. Nahrstedt, "MON : On-demand Overlays for Distributed System Management", *Proceedings of the 2nd conference on Real, Large Distributed Systems*, WORLDS, pp. 13–18, Berkely, CA, USA, 2005, Usenix Association.

[110] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal Multicast", *ACM Transactions on Computer Systems*, 17:41–88, ACM, 1999.

[111] Y. Chu, S. Rao, S. Seshan, and H. Zhang, "A case for end system multicast", *IEEE Journal on Selected Areas in Communications*, 20:1456–1471, IEEE, 2002.

[112] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance", *PODC'87: Proceedings of the 6th Symposium on Principles of Distributed Computing*, pp. 1–12, ACM, 1987.

[113] A. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Scamp: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication", *Networked Group Communication*, Lecture Notes in Computer Science, pp. 44–55, Springer Berlin / Heidelberg, 2001.

[114] J. Leitão, J. Pereira, and L. Rodrigues, "HyParView: A membership protocol for reliable gossip-based broadcast", *DSN'07: Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 419–429, IEEE, 2007.

[115] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store", *SIGOPS Operating Systems Review*, 41:205–220, ACM, 2007.

[116] J. Liu and M. Zhou, "Tree-assisted gossiping for overlay video distribution", *Multimedia Tools and Applications*, 29:211–232, Kluwer Academic Publishers, 2006.

[117] S. Voulgaris, M. Jelasity, and M. V. Steen, "A Robust and Scalable Peer-to-Peer Gossiping Protocol", *Agents and Peer-to-Peer Computing*, volume 2872 of *Lecture Notes in Computer Science*, pp. 47–58, Springer, 2005.

[118] R. Melamed and I. Keidar, "Araneola: A scalable reliable multicast system for dynamic environments", *Journal of Parallel and Distributed Computing*, 68(12):1539–1560, Academic Press, Inc., 2008.

[119] B. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, 13:422–426, ACM, 1970.

[120] B. Kaldehofe, "Buffer management in probabilistic peer-to-peer communication protocols", *SRDS'03: Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pp. 76–85, IEEE, 2003.

[121] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems", *Middleware'01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pp. 329–350, Springer-Verlag, 2001.

[122] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: topic-based event routing for peer-to-peer architectures", *DEBS '07: Proceedings of the 1st International Conference on Distributed event-based systems*, pp. 2–13, New York, NY, USA, June 2007, ACM Press.

[123] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast", *ICNP'06: Proceedings of the 14th International Conference on Network Protocols*, pp. 2–11, IEEE, 2006.

[124] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh", *SOSP'03: Proceedings of the 19th Symposium on Operating Systems Principles*, pp. 282–297, ACM, 2003.

[125] J. A. Patel, E. Rivière, I. Gupta, and A.-M. Kermarrec, "Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 53:2304–2320, Elsevier North-Holland, Inc., 2009.

[126] S. Voulgaris and M. Van Steen, "Hybrid Dissemination: Adding Determinism to Probabilistic Multicasting in Large-scale P2P Systems", *Middleware'07: Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, pp. 389–409, Springer-Verlag, 2007.

[127] Z. Li, G. Xie, K. Hwang, and Z. Li, "Churn-Resilient Protocol for Massive Data Dissemination in P2P Networks", *IEEE Transactions on Parallel and Distributed Systems*, 22:1342–1349, IEEE Computer, 2011.

[128] Z. Li, G. Xie, and Z. Li, "Towards reliable and efficient data dissemination in heterogeneous peer-to-peer systems", *IPDPS'08: Proceedings of the 22th International Parallel and Distributed Processing Symposium*, pp. 1–12, IEEE, 2008.

[129] Z. Fei and M. Yang, "A proactive tree recovery mechanism for resilient overlay multicast", *IEEE/ACM Transactions on Networking*, 15:173–186, IEEE Press, 2007.

[130] C. Tang, R. N. Chang, and C. Ward, "GoCast: Gossip-Enhanced Overlay Multicast for Fast and Dependable Group Communication", *DSN'05: Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 140–149, IEEE, 2005.

[131] J. Leitão, J. Pereira, and L. Rodrigues, "Epidemic Broadcast Trees", *SRDS'07: Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pp. 301–310, Beijing, China, 2007, IEEE.

[132] M. Ferreira, J. Leitao, and L. Rodrigues, "Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay", *SRDS'10: Proceedings of the 29th International Symposium on Reliable Distributed Systems*, pp. 293–302, IEEE, 2010.

[133] M. Matos, V. Schiavoni, P. Felber, R. Oliveira, and E. Rivière, "Lightweight, efficient, robust epidemic dissemination", *Journal of Parallel and Distributed Computing*, 73(7):987999, 2013.

[134] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. van Steen, "Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks", *IPTPS'06: Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006.

[135] R. van Renesse, K. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining", *ACM Transactions on Computer Systems*, 21:164–206, ACM, 2003.

[136] E. Rivière, R. Baldoni, H. Li, and J. Pereira, "Compositional gossip", *ACM SIGOPS Operating Systems Review*, 41(5):43, ACM, October 2007.

[137] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live", *PODC'07: Proceedings of the 26th Symposium on Principles of Distributed Computing*, pp. 398–407, ACM, August 2007.

[138] F. Maia, M. Matos, J. Pereira, and R. Oliveira, "Worldwide consensus", *DAIS'11: Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems*, pp. 257–269, Springer-Verlag, 2011.

[139] S. Voulgaris, *Epidemic-Based Self-Organization in Peer-to-Peer Systems*, Ph.D. Dissertation, Vrije Universiteit, Amsterdam, November 2006.

[140] A. Stavrou, D. Rubenstein, and S. Sahu, "A lightweight, robust p2p system to handle flash crowds", *IEEE Journal on Selected Areas in Communications*, 2002.

[141] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, "Practical, Distributed Network Coordinates", *HotNets'03: Proceedings of the 2nd Workshop on Hot Topics in Networks*, ACM, 2003.

[142] O. Beaumont, A.-M. Kermarrec, and E. Rivière, "Peer to peer multidimensional overlays: Approximating complex structures", *OPODIS'07: Proceedings of the 11th International Conference On Principles Of Distributed Systems*, Lecture Notes in Computer Science, December 2007.

[143] O. Beaumont, A.-M. Kermarrec, L. Marchal, and É. Rivière, "VoroNet: A scalable object network based on Voronoi tessellations", *IPDPS'07: Proceedings of the 21st International Parallel and Distributed Processing Symposium*, IEEE, 2007.

[144] Y. Wang and X.-Y. Li, "Distributed Spanner with Bounded Degree for Wireless Ad Hoc Networks", *IPDPS'02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IEEE, 2002.

[145] A. C.-C. Yao, "On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems", *SIAM Journal on Computing*, 11(4):721–736, 1982.

[146] L. Barba, P. Bose, M. Damian, R. Fagerberg, W. L. Keng, J. O'Rourke, A. van Renssen, P. Taslakian, S. Verdonschot, and G. Xia, "New and Improved Spanning Ratios for Yao Graphs", *SoCG'14: 30th Annual Symposium on Computational Geometry*, ACM, 2014.

[147] I. Kanj and G. Xia, "On certain geometric properties of the Yao-Yao graphs", *Journal of Combinatorial Optimization*, pp. 1–10, Springer US, 2012.

[148] Y. Liu, Y. Guo, and C. Liang, "A survey on peer-to-peer video streaming systems", *Peer-to-Peer Networking and Applications*, 1:18–28, 2008.

[149] O. Abboud, K. Pussep, A. Kovacevic, K. Mohr, S. Kaune, and R. Steinmetz, "Enabling resilient P2P video streaming: survey and analysis", *Multimedia Systems*, 17(3):177–197, 2011.

[150] A. Barrios, M. Barrios, D. De Vera, P. Rodríguez-Bocca, and C. Rostagnol, "GoalBit: a free and open source peer-to-peer streaming network", *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pp. 727–730, New York, NY, USA, 2011, ACM.

[151] X. Zhang, J. Liu, B. Li, and Y.-S. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming", *INFOCOM'05: Proceedings of the 24th Annual Joint Conference of the Computer and Communications Societies*, volume 3, pp. 2102 – 2111 vol. 3, IEEE, 2005.

[152] D. Frey, R. Guerraoui, A.-M. Kermarrec, and M. Monod, "Boosting Gossip for Live Streaming", *P2P'10: Proceedings of the 10th International Conference on Peer-to-Peer Computing*, pp. 1–10, 2010.

[153] H. Chang, S. Jamin, and W. Wang, "Live Streaming With Receiver-Based Peer-Division Multiplexing", *IEEE/ACM Transactions on Networking*, 19(1):55–68, February 2011.

[154] F. Wang, Y. Xiong, and J. Liu, "mTreebone: A Collaborative Tree-Mesh Overlay Network for Multicast Video Streaming", *IEEE Transactions on Parallel and Distributed Systems*, 21(3):379–392, March 2010.

[155] F. Picconi and L. Massoulié, "Is there a future for mesh-based live video streaming?", *P2P'08: Proceedings of the Eighth International Conference on Peer-to-Peer Computing*, pp. 289–298, IEEE, 2008.

[156] S. J. Wee and J. G. Apostolopoulos, "Secure scalable streaming enabling transcoding without decryption", *ICIP'01: Proceedings of the 2001 International Conference on Image Processing*, volume 1, pp. 437–440, IEEE, 2001.

[157] B. Yu and B. Xiao, "Detecting selective forwarding attacks in wireless sensor networks", *IPDPS'06: Proceedings of the 20th International Parallel and Distributed Processing Symposium*, pp. 8–pp, IEEE, 2006.

[158] I. Krontiris, T. Dimitriou, T. Giannetsos, and M. Mpasoukos, "Intrusion detection of sinkhole attacks in wireless sensor networks", *Algorithmic Aspects of Wireless Sensor Networks*, pp. 150–161, Springer, 2008.

[159] T. H. Hai and E.-N. Huh, "Detecting selective forwarding attacks in wireless sensor networks using two-hops neighbor knowledge", *NCA'08: Proceedings of the 7h International Symposium on Network Computing and Applications*, pp. 325–331, IEEE, 2008.

[160] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for Web Transactions", *ACM Transactions on Information and System Security*, 1(1):66–92, ACM, New York, NY, USA, November 1998.

[161] P. Garbacki, A. Iosup, D. Epema, and M. van Steen, "2Fast: Collaborative Downloads in P2P Networks", *P2P'06: Proceedings of the 6th International Conference on Peer-to-Peer Computing*, pp. 23–30, IEEE, 2006.

[162] A. Noureddine, R. Rouvoy, and L. Seinturier, "A review of energy measurement approaches", *ACM SIGOPS Operating Systems Review*, 47(3):42–49, ACM, 2013.