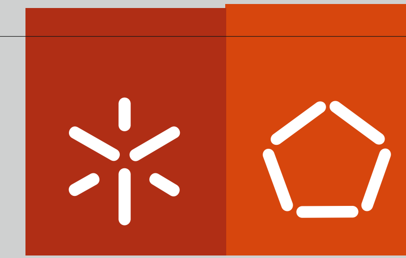
Epidemic Algorithms for Large Scale Data Dissemination
Miguel Ângelo Marques de Matos
UMinho|2013

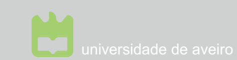


Universidade do Minho
Escola de Engenharia

Miguel Ângelo Marques de Matos

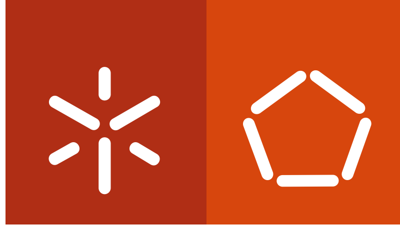
Epidemic Algorithms for Large Scale Data Dissemination

Programa de Doutoramento em Informática
das Universidades do Minho, de Aveiro e do Porto



Universidade do Minho

Julho de 2013

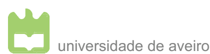


Universidade do Minho
Escola de Engenharia

Miguel Ângelo Marques de Matos

Epidemic Algorithms for Large Scale Data Dissemination

Programa de Doutoramento em Informática
das Universidades do Minho, de Aveiro e do Porto



Universidade do Minho

Trabalho realizado sob a orientação do
Professor Doutor Rui Carlos Oliveira

Julho de 2013

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE:

Universidade do Minho, 30 / 07 / 2013

Assinatura: ____

Agradecimentos

Todas as viagens começam por um pequeno passo e transformam-se progressivamente num conjunto imensurável de passos. Tive a sorte de, ao longo deste percurso, ser sempre acompanhado por pessoas fantásticas que o tornaram possível. Um obrigado a todos, pois cada passo, grande ou pequeno, que demos em conjunto levou-me até aqui e está indelevelmente presente neste trabalho.

Em primeiro lugar, gostaria de agradecer ao meu orientador, Prof. Rui Oliveira, pela incansável paciência, dedicação e orientação, dentro e fora do contexto deste trabalho, que excedeu largamente o profissionalmente exigido. Foi, e é, um privilégio poder trabalhar consigo.

Não poderia deixar de agradecer também ao restante corpo docente do Grupo (com G maiúsculo) de Sistemas Distribuídos que, apesar da dimensão, sabe estar coeso e sempre disponível para ajudar e festejar. Uma palavra de apreço para o Prof. José Pereira pela disponibilidade e interesse constante em auxiliar-me.

Este caminho teria sido muito mais difícil sem o excelente ambiente que se vive no laboratório 2.20. Nunca, ao longo dos últimos sete anos que lá estou, me senti colocado de lado ou desrespeitado e sempre que necessário, tantas vezes sem necessitar de pedir, surgiu uma mão amiga pronta a auxiliar no trabalho ou a participar num momento de lazer. Isto é fruto das pessoas excelentes que lá estão e que por lá passaram: Alfrânio Correia, Ana Nunes, Bruno Costa, Daniel Machado, Filipe Campos, Francisco Cruz, Francisco Maia, Luís Ferreira, Luís Soares, João Paulo, José Marques, Miguel Borges, Nelson Gonçalves, Nuno Carvalho, Nuno Castro, Nuno Lopes, Paulo Jesus, Pedro Gomes, Ricardo Gonçalves e Ricardo Vilaça. Não posso deixar de mencionar também os membros d'Os Sem Estatuto pelo companheirismo e diversão proporcionada e em particular ao Jácome Cunha pela boa disposição e disponibilidade.

I was also lucky enough to substantially collaborate throughout most of this dissertation with the people at the University of Neuchâtel, Switzerland, namely Pascal Felber, Etienne Rivière and Valerio Schiavoni. I learned a lot from you and I have become, I expect, a better researcher and a better person in the process. Thank you very much, I expect we can continue to work together.

Não seria quem sou hoje se não fosse pelos amigos do Tour com os quais fiz todo o percurso de licenciatura e me acompanharam em ínumeros momentos difíceis: Agostinho Silva, André Rodrigues, Avelino Rego, Duarte Alves, Nelson Silva, Paulo Sousa, Rui Ribeiro e Vítor Rocha. Além dos trabalhos feitos em conjunto e de todas as borgas, ficou uma amizade para a vida, grande abraço!

A minha família é o pilar sobre o qual tudo assenta. Nada disto seria possível, e mesmo sendo não teria significado, sem a sua presença constante. Em particular da minha mãe e pai, Nino, Iara, Xana e Pedro. Sou um felizardo por poder contar com vocês.

Finalmente à Ana, que partilha comigo todos os momentos e está sempre presente. Na ausência de palavras, só desejo fazer-te tão feliz como me fazes a mim.

Adicionalmente, algumas instituições apoiaram o trabalho apresentado nesta tese. A Fundação para a Ciência e Tecnologia (FCT) apoiou este trabalho através da bolsa de doutoramento (SFRH / BD / 62380 / 2009). O Departamento de Informática da Univesidade do Minho e o HASLab - High Assurance Software Lab. ofereceram-me as condições necessárias para desenvolver o trabalho conducente a esta tese.

Braga, Julho de 2013
Miguel Matos

To Anne

Epidemic Algorithms for Large Scale Data Dissemination

Distributed systems lie at the core of modern IT infrastructures and services, such as the Internet, e-commerce, the stock exchange, Cloud Computing and the SmartGrid. These systems, built and developed throughout the last decades, have relied, due to their importance, on distributed algorithms with strong correctness and safety guarantees. However, such algorithms have failed to accompany, for theoretical and practical reasons, the requirements of the distributed systems they support in terms of scale, scope and pervasiveness. Reality is unforgiving and thus researchers had the need to design and develop new algorithms based on probabilistic principles that, despite their probabilistic yet quantifiable guarantees, are suitable to today's modern distributed systems.

In this dissertation, we study the challenges of and propose solutions for, applying probabilistic dissemination algorithms, also known as epidemic- or gossip-based, in very large scale distributed systems. In particular, we focus on the issues of scalability of content types (topic-based publish-subscribe), content size (efficient data dissemination) and ordering requirements (total order). For each one of these issues, we present a novel distributed algorithm that solves the problem while matching state-of-the art performance and trade-offs, and evaluate it on a realistic setting.

Algoritmos Epidémicos para Disseminação de Dados em Larga Escala

Os sistemas distribuídos são hoje em dia componentes fundamentais das infraestruturas e serviços de TI, tais como a Internet, comércio eletrónico, a bolsa, Computação em Nuvem (*Cloud Computing*) e a *SmartGrid*. Dada a sua importância, estes sistemas, desenhados e desenvolvidos ao longo das últimas decadas, assentam em algoritmos distribuídos com garantias fortes de correção e segurança (*safety*). No entanto, esses algoritmos têm ficado para trás por razões teóricas e práticas, no que concerne às necessidades de escala, âmbito e universalidade dos sistemas distribuídos que suportam. Este hiato conduziu à conceção e desenvolvimento de novos algoritmos baseados em princípios probabilísticos que, apesar de oferecerem somente garantias probabilísticas ainda que quantificáveis, conseguem responder às necessidades dos sistemas distribuídos modernos.

Esta dissertação estuda os desafios e propõe soluções para a utilização de algoritmos probabilisticos de disseminação de dados, também conhecidos como epidémicos ou baseados em rumor, em sistemas distribuídos de larga escala. Foca-se, em particular, na escalabilidade de tipos de conteúdo (paradigma publicação-subscrição baseado em tópicos), tamanho do conteúdo (disseminação eficiente de dados) e requisitos de ordem (ordem total). Para cada problema identificado acima, propomos um novo algoritmo distribuído que o resolve, sem prejuízo dos compromissos oferecidos por outros algoritmos estado da arte, e avaliamos a solução num ambiente realista.

Contents

1 Introduction **1**

1.1 Problem statement and objectives . . . 3
1.2 Contributions . . . 8
1.3 Results . . . 9
1.4 Dissertation outline . . . 12

2 Background **13**

2.1 Model . . . 13
2.2 Overlay Networks . . . 14
2.2.1 Structured Overlays . . . 15
2.2.2 Unstructured Overlays . . . 16
2.2.3 Discussion . . . 18
2.3 Data Dissemination . . . 19
2.3.1 Flooding . . . 19
2.3.2 Tree . . . 19
2.3.3 Epidemic . . . 20
2.3.4 Discussion . . . 22
2.4 Conventions . . . 22

3 StaN: scalable topic-based publish-subscribe **25**

3.1 Introduction . . . 25
3.2 Algorithm description . . . 27
3.2.1 System Model and Assumptions . . . 27
3.2.2 Design Rationale . . . 29
3.2.3 Link Management . . . 31
3.2.4 Dissemination . . . 34

3.3 Evaluation 36
3.3.1 Experimental Data 37
3.3.2 Workload Characteristics 38
3.3.3 Experimental Setup 41
3.3.4 Performance 42
3.3.5 Fitness 46
3.3.6 Dynamics 48
3.3.7 Greedy-omniscient comparison 51
3.3.8 Dissemination 53
3.4 Related Work 55
3.5 Discussion 58

4 Brisa: efficient reliable data dissemination 61
4.1 Introduction 61
4.2 Algorithm description 64
4.2.1 Peer Sampling Service Layer 64
4.2.2 Rationale 66
4.2.3 Emergence of a Dissemination Structure 68
4.2.4 Preventing Cycles 69
4.2.5 Parent Selection Strategies 70
4.2.6 Dynamism 71
4.2.7 Generalized Dissemination Structures 73
4.2.8 Multiple Dissemination Structures 74
4.3 Evaluation 76
4.3.1 Structural properties 77
4.3.2 Network properties 79
4.3.3 Robustness 81
4.3.4 Multiple trees 82
4.3.5 Comparison with existing approaches 86
4.4 Related Work 91
4.5 Discussion 96

5 EpTO: epidemic total order dissemination 99
5.1 Introduction 99
5.2 Algorithm Description 101

5.2.1 System model and assumptions 102
5.2.2 Problem Statement 102
5.2.3 Rationale 103
5.2.4 Detailed description 105
5.2.5 Deliverability oracle and logical time 108
5.2.6 Properties satisfiability 111
5.3 Evaluation 120
5.4 Related Work 127
5.5 Discussion 128

6 Conclusions 131
6.1 Future work 134

Bibliography 137

List of Figures

1.1 Epidemic dissemination problems 5

3.1 StaN placement in the problem space. 26

3.2 StaN's architecture. 29

3.3 5-processes sample run with two topics from the point of view of process n_0 (only a subset of the links is shown). 34

3.4 Subscription distribution for LiveJournal universes. 39

3.5 Subscription distribution for Wikipedia universes. 40

3.6 Subscription correlation. 40

3.7 Evolution and distribution of the LVS and PVS for a synthetic universe (100 processes and 16 topics). The legend is shared by the two graphs. 43

3.8 IPVS/FPVS and relative improvement for the $\mathcal{L}_8$ and $\mathcal{W}_8$ universes. 44

3.9 Evolution of the IPVS/FPVS with the number of topics. 45

3.10 Clustering coefficient, average path length and diameter distribution for the $\mathcal{L}_8$ and $\mathcal{W}_8$ universes (most lines overlap). 47

3.11 View evolution under message loss for universe $\mathcal{W}_8$ (percentages indicate message loss rates). 48

3.12 Universe and view evolution under churn for universe $\mathcal{W}_8$. (Numbers represent the churn speedup factor.) 50

3.13 View evolution for growing $\mathcal{W}_8$ universe. 51

3.14 Comparison of view improvement and clustering coefficient distribution for StaN and a greedy-omniscient approach for $\mathcal{W}_8$. Lines "Initial" and "Final StaN" overlap in Figure 3.14(b). 52

3.15 SimpleFlood vs CrosspostFlood on universe $\mathcal{L}_8$: a) bandwidth reduction before and after optimizing the overlays with StaN b) hops necessary for first delivery. 54

4.1 Brisa placement in the problem space. 62
4.2 HyParView (Leitão et al. 2007b): views maintenance. 64
4.3 Distribution of duplicates per message for each process for 500 messages in a 512 processes HyParView network for various active view sizes. 66
4.4 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 process sending the duplicate. 69
4.5 Avoiding creating a cycle for a tree, by checking that process N is not in the dissemination path to the potential parent. 70
4.6 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 process. 73
4.7 Depth distribution for 512 process (first-come first-picked strategy). 77
4.8 Degree distribution for 512 process (first-come first-picked strategy). 77
4.9 Sample tree shape for 100 processes represented in a radial layout. The HyParView active view size of 4 (left) and 8 (right). Expansion factor is 1. 78
4.10 Routing delays distribution on PlanetLab for 150 processes. Structure is a tree with view size 4. Message size is 1KB×200 messages. 80
4.11 Download bandwidth usage for 512 processes. 80
4.12 Upload bandwidth usage for 512 processes. 81
4.13 Distribution of the number of trees where processes are leaves for 512 processes and active view size of 8. 83
4.14 Distribution of the number of children across all trees for 512 processes and active view size of 8. 84
4.15 Reception delays per message when using multiple trees for 512 processes and active view size of 8. The number of messages is 500. 85
4.16 Dissemination delay when splitting the stream of messages across multiple trees for 512 processes and active view size of 8. The number of messages is 500. 86

4.17 Comparison of bandwidth usage for 512 processes. 89
4.18 Construction time for 512 (on cluster) and 200 (PlanetLab) processes. X axis is logarithmic. 89
4.19 Parent recovery delays for 128 processes with active view size 4 under 3% continuous churn. 91

5.1 EpTO placement in the problem space. 100
5.2 Properties satisfiability: order but no agreement (left) and agreement but no order (right). 102
5.3 Totally ordered event delivery. 103
5.4 EpTO architecture. 105
5.5 Stability oracles. 108
5.6 Sample run with a logical clock. Note that rounds are labeled just for presentation purposes, EpTO does not require round synchronization or labeling. 110
5.7 Latency distribution used in experiments (obtained from PlanetLab). 120
5.8 Spontaneous order with a global and a logical clock for varying system sizes and publication rates r. 121
5.9 Delivery delay for 100, 200 and 500 processes for varying publication rates r and clock types. 122
5.10 Pathological disorder situations related to the task execution period δ for a publication rate $r = 0.5$. 123
5.11 Impact of churn on the delivery delay with a global clock and publication rate $r = 0.5$. 126

6.1 Placement of each proposed algorithm in the problem space. . . . 132
6.2 Possible problem combinations and challenges. 135

List of Tables

3.1 Universe configurations. 38

4.1 Impact of churn for a 128 and 512 node networks with active view size 4. 83
4.2 Protocol design space. 86
4.3 Dissemination latency for 512 processes for 500 messages of 1KB. 90

Chapter 1

Introduction

Since the dawn of time, our species has distinguished itself from the others by its superior communication capabilities and the ability to cooperate toward a common goal. Fast forwarding to the present days, our Information society is still characterized by its blazingly fast communication capabilities and the ability to cooperate worldwide by quickly exchanging information among several parties geographically dispersed, from text and voice to stock information and video. This ability rests on Telecommunication Technology and Information Technology and forms the backbone of infrastructures as critical as electricity distribution and the trend toward the Smartgrid, the Internet and Cloud Computing, and useful services such as social networks and video broadcasting platforms. The impressive growth in the number of infrastructures, machines and people interconnected together resulted also on the exponential growth of data that flows throughout these systems. As a matter of fact, recent studies estimate that the amount of data available quadruples every eighteen months (Gantz 2007, 2008) making *Big Data* a hot research and business topic. Remarkably, when discussing all these subjects and systems, and the challenges they raise, we inevitably come down to a well-known subject: distributed computing systems. At its inception, distributed systems address essentially two problems: scalability - the capacity of a single component is not enough to cope with the demand and thus there is the need to distribute that load to several components; and/or availability - the service disruption caused by the failure of a single component is unacceptable and thus the component needs to be replicated as a means to fault tolerance.

Conceptually, distributed systems are composed of interconnected processes

that execute computing instructions concurrently and independently, and may possess only limited information about the system as a whole (Lynch 1996). Depending on the environment they are able to operate, distributed systems can be classified according to their timing model - synchronous or asynchronous -, interprocess communication method - shared memory or message passing - , and failure model - well-defined or arbitrary failures (Lynch 1996). In this dissertation we are interested in asynchronous, message passing system with well defined failure models regarding processes and communication channels.

Until recently, the bulk of research on distributed systems focused on the design and implementation of algorithms with strong guarantees even in the presence of process and communication failures without assumptions on processes and communication relative speed. Examples include agreement (Guerraoui and Schiper 2001), robust dissemination (Floyd et al. 1997) or leader election (Sabel and Marzullo 1995). However, the huge growth in terms of number of processes and communication complexity observed in the last decade rendered classical algorithms impractical for systems encompassing hundreds to thousands of processes. The reason for this stems not only from the deterministic nature of these algorithms, which poses well-known restrictions on what can be achieved in a distributed system (Fischer et al. 1985), but also on the properties that are achievable at a large scale and their cost. In fact, the well-known CAP theorem (Brewer 2000; Gilbert and Lynch 2002) states that one cannot achieve at the same time consistency, availability, and partition tolerance. As failure rates increase with the system size (Schroeder and Gibson 2007; Schroeder et al. 2009; Verespej and Pasquale 2011), the larger the system the harder and costlier those properties are to achieve. Besides, algorithms offering strong guarantees cope poorly with the scale of modern distributed systems (Demers et al. 1987; Birman et al. 1999; Vogels 2009), thus leaving a gap between what is expected by an application architect and what is achievable in practice.

As an alternative to the dilemma posed by deterministic algorithms, a lot of research was dedicated to probabilistic algorithms instead. Remarkably, by sidestepping the deterministic decisions taken by classical algorithms and using randomization instead, probabilistic algorithms become surprisingly robust to failures (Demers et al. 1987; Birman et al. 1999), scalable (Demers et al. 1987; Birman et al. 1999) and able to overcome impossibility results of deterministic al-

gorithms (Ben-Or 1983). The probabilistic, yet quantifiable, guarantees of these algorithms have proved sufficient to address most problems in distributed systems, such as membership management (Ganesh et al. 2001; Jelasity et al. 2007b; Leitão et al. 2007b), failure detection services (Renesse et al. 2007), robust dissemination (Birman et al. 1999; Carvalho et al. 2007), leader election (Gupta et al. 2000), or indexing mechanisms (Montresor et al. 2005; DeCandia et al. 2007b). When the randomization happens at the interprocess communication level, the message exchange patterns among processes actually resembles the spreading of a rumor or epidemic. Thus such algorithms are known as gossip or epidemic (Bailey 1975; Demers et al. 1987). As a matter of fact, the way messages are exchanged and the guarantees epidemic algorithms offer on such exchanges is a crucial design axis of epidemic systems. In general, in a message passing system, regardless of the properties of the underlying communication channels, we need to consider several fundamental aspects: delivery reliability, message type, message size, latency and ordering.

When properly configured, an epidemic algorithm ensures delivery reliability to all nodes, *with high probability*. The notion of *with high probability*, shortened as w.h.p. quantifies the probability of a given property holding, such as *all processes receive all broadcast messages* and is usually given by $1 - \epsilon$, ϵ being an arbitrarily small quantity (Birman et al. 1999; Eugster et al. 2003b). On top of these reliability guarantees, it is possible to build other algorithms that address each one of the concerns pointed above and model several real world scenarios. In the next sections, we outline each one of these concerns, present the existing problems and frame the contributions of this dissertation.

1.1 Problem statement and objectives

Given the inadequacy of classical deterministic algorithms in addressing today's problems of scale and cost, and the consequent issues of obtaining strong deterministic guarantees in highly dynamic environments, we started to look at epidemic algorithms as a promising alternative. As a matter of fact, epidemic algorithms have been used to address many distributed systems problems (see for instance (Rivière and Voulgaris 2011) for a recent survey) not just on the research community but also on industrial environments such as Amazon's Dy-

namo (DeCandia et al. 2007a) and Facebook's Cassandra (Lakshman and Malik 2010), which use epidemic algorithms in key parts of their systems.

The general research question pervading this dissertation is thus:

> What key weaknesses preclude epidemic algorithms from being broadly applied to a wider range of very large scale scenarios?

Instead of attacking the question in a top-down approach and addressing a particular research problem, such as *epidemic algorithms impose an excessive overhead in the network*, we followed a bottom-up approach instead by uncovering the key aspects leading to these problems, such as *careless excessive transmissions of large messages result in bandwidth depletion.* Because the transmission of messages is the defining property of a message passing system in general and of epidemic algorithms in particular, the three challenges we address in this dissertation are concerned with the properties of the transmission and the properties of the messages being transmitted. More precisely, the research questions studied in this dissertation are:

1. How can we deal with different message types and what is the impact on management overhead?
2. How can we deal with large message sizes and what is the impact on bandwidth and latency?
3. How can we deal with message ordering, and in particular total order?

Note that each one of these questions have been already addressed using classical, often centralized deterministic algorithms. For instance, the problems of message types has been addressed by the use of publish-subscribe systems based on centralized brokers (Carzaniga et al. 2000; Castro et al. 2002), the dissemination of large messages is addressed using traditional tree construction mechanisms (Chu et al. 2002) or algorithms such as BitTorrent (Cohen 2003, 2008), and message ordering is a very well studied subject (Défago et al. 2004). Our concern here is to build scalable algorithms suitable to the size of modern distributed systems that are able to operate on dynamic environments subject to recurring faults and churn - a consequence of scale itself. The scope of our problems is depicted in Figure 1.1. In the remainder of this section we briefly discuss each one of the problems.

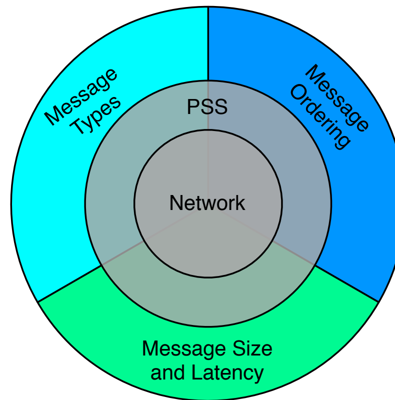


Figure 1.1: Epidemic dissemination problems

Message types The ability to consider different message types allows not only to differentiate between control and application level messages but also to convey a richer semantic meaning for the application itself. For example, by mapping a given message type to a specific kind of information - a topic - we are able to model a system where processes might only be interested in specific topics. Processes are then able to publish messages in a given topic and subscribe to topics they are interested in and thus form a topic-based publish-subscribe system (Eugster et al. 2003a). Despite dismayingly simple, this abstraction fully captures the behavior of many real scenarios such as Usenet, Web syndication and social networks. The literature on topic-based publish-subscribe is rich and extensive and addresses problems such as topic construction, inter- and intra-topic message dissemination and scalability to very large number of processes (Baldoni et al. 2007a; Chockler et al. 2007a,b; Girdzijauskas et al. 2010). However, the way we produce and consume data nowadays is departing from the traditional model of one-producer to many-consumers that can be observed in a typical newspaper to a new paradigm where everyone is simultaneously a producer and consumer of information. This is true not only for common users with the advent of the Web 2.0 and tools such as blogs, but also for enterprises where data needs to flow to and from several locations to support globalized businesses. On a distributed system this results not only on a sheer increase in the number of processes in the system

as well as on the number of topics available. To the best of our knowledge, existing approaches fall short on managing very large number of topics by incurring on excessive overhead, by degrading the properties that make epidemic dissemination robust or both. Therefore, we strive for a topic-based publish-subscribe algorithm that addresses the following challenges: 1) scalability in the number of processes, 2) scalability in the number of topics, 3) fitness and robustness to epidemic dissemination, 3) completeness by having processes receive all messages published in their subscribed topics, and 4) accuracy by avoiding that processes receive messages they are not interested in. Part of our solution to this problem rests on a social observation: topic subscribers often share similar interests and thus one can leverage on this topic overlap as a means to achieve topic scalability. The design, implementation and evaluation of this algorithm is presented in Chapter 3.

Message size and latency Without loss of generality, we can see data dissemination as the need to distribute arbitrarily large data (contents) from a set of nodes that hold the data to a very large population of nodes that demand it. The data to be disseminated could range from updates or new versions of a popular software, such as an operating system, to the transmission of live sport events, both of which have very different requirements. For instance, in the former, data integrity is a major concern, while in the latter jitter tends to be more relevant. To enable large scale data dissemination, both in terms of number of nodes and volume of data, several approaches emerged that focus either on the optimization of the data sources (the nodes that hold the data) or in the exploitation of the scale and characteristics of the target nodes, i.e. the nodes that demand the data. The typical example of the first approach is Content Distribution/Delivery Networks, such as Akamai (Akamai Technologies 2013), that essentially mirror the data sources across geographically dispersed locations. In this way consumer nodes can retrieve the data from closer locations thus improving performance, while reducing the load on the original data sources. The other canonical approach is to leverage on the nodes demanding the data and have them behave in a cooperatively manner by exchanging data with their peers. The most popular peer-to-peer system used nowadays is probably BitTorrent (Cohen 2003, 2008).

Despite the popularity and the necessity of large scale data dissemination

systems, there are still outstanding issues to solve. Existing epidemic systems are able to scale in the number of nodes and offer probabilistic delivery guarantees (Kermarrec et al. 2001; Ganesh et al. 2001; Eugster et al. 2004). Nonetheless, they impose a heavy load on the network that precludes its usage for disseminating large amounts of data. Based on this, several proposals address the problem of heavy bandwidth consumption by using designs that are more efficient (Castro et al. 2003b; Locher et al. 2007). However, efficiency here comes at the price of fault and churn tolerance which is essential in very-large scale systems, where the population of nodes could not be expected to remain stable (Schroeder and Gibson 2007; Schroeder et al. 2009; Verespej and Pasquale 2011). With these problems and limitations in mind, we envision a very large scale data management system, with a particular emphasis on data dissemination. Thus, our goal is to build an epidemic data dissemination that addresses the following challenges: 1) scalability in the number of processes, 2) robustness under faults and churn, 3) efficient bandwidth usage, and 4) adjustable to heterogeneous environments. We approach this problem in two steps. First, we rely on the robustness and scalability, yet inefficient, properties of epidemic dissemination as a safety net. Then, we increase the efficiency by observing that it is the possibility of receiving duplicates, not the actual reception, that makes epidemic algorithms so robust. The design, implementation and evaluation of an algorithm addressing these challenges is presented in Chapter 4.

Message ordering The order of events, carried in messages, is one of the most fundamental problems in distributed systems (Lamport 1978) and as such a very large body of knowledge has been dedicated to it (Défago et al. 2004). The ability to order events in the same way irrespective of the size of the system, relative processor speeds, failures and channel asynchrony on distinct processes is extremely powerful and greatly simplifies the design and implementation of a wide range of related algorithms, such as state machine replication, view synchrony and consensus. Intuitively, the ordering of events is concerned with the sequence of events that processes are allowed to deliver to an application. While this is trivial to achieve in a system where just a single process broadcasts events - by tagging each event with a sequence number - it becomes rather complex and costly when more than one process may broadcast events. For instance, suppose

two operations such as add and multiplication where the order they are applied matters, i.e. they are not commutative. Now suppose one broadcasts two events in a distributed system where one event carries the operation of adding a quantity to a bank account and the other event carries the operation of applying an interest rate to that account balance. Clearly, for the system to remain semantically correct and coherent, the order in which these two events are applied in *all* processes needs to be the same. Different applications might live with different ordering guarantees related to the senders, such as FIFO and causal order, the receivers, such as total order, or a combination of both. Because senders can easily encode ordering constraints in the events they broadcast, for instance by using sequence numbers in the case of FIFO or defining the *happens before* relationship in the case of causal order (Lamport 1978), we focus instead on enforcing order at the receivers, i.e. total order. Due to the strong abstraction they provide, total order algorithms are complex to design, hard to implement, scale poorly and are sensitive to failures (Felber and Pedone 2002; Cimmino et al. 2003; Défago et al. 2004). Therefore, we seek an epidemic total order algorithm that addresses the following challenges: 1) ensures total order, 2) scales with the number of processes and events, and 3) is robust to failures and churn. To this end, we start with an epidemic dissemination algorithm with well-known reliability guarantees and build the notion of stability and ordering on top of it. The design, implementation and evaluation of an algorithm addressing these challenges is presented in Chapter 5.

1.2 Contributions

The main contributions of this dissertation are:

- StaN, a new topic-based publish-subscribe algorithm that addresses the problem of message types by being able to scale on the number of processes and topics, while preserving the fitness and robustness of epidemic dissemination algorithms. Completeness and accuracy are obtained by design by managing each topic independently. Moreover, we also devise a dissemination algorithm that takes advantage of crossposting, i.e. events simultaneously published to several topics, to reduce bandwidth usage without compromising latency.

- Brisa, a new dissemination algorithm combining the robustness and scalability of epidemic approaches with the resource efficiency of structured solutions. It supports multi-source dissemination with minimal overhead by reusing the dissemination structures whenever possible. These dissemination structures, which can be either trees or directed acyclic graphs, are built in a completely decentralized manner and can be adjusted to different optimal criteria, such as end-to-end latency and heterogeneous process capacities.

- EpTO, a new total order algorithm able to scale to a very large number of processes and events. Agreement is probabilistic but all the other properties, namely validity, integrity and total order are deterministic and always preserved. The algorithm is shown to be very robust to churn and misconfiguration with performance degrading only under the most adverse environments.

1.3 Results

The work conducted during this dissertation has been published in several conference and journal papers. In chronological order, these are:

- StaN: Exploiting shared interests without disclosing them in gossip-based publish/subscribe.
  Miguel Matos, Ana Nunes, Rui Oliveira, and José Pereira.
  In International Workshop on Peer-to-Peer Systems (IPTPS), 2010.

  This paper presents the preliminary idea leading to StaN with an evaluation of the algorithm in a synthetic trace. The major result is the recognition that it is possible to manipulate the topics without affecting the organizational properties necessary for robustness.

- Brisa: Combining Efficiency and Reliability in Epidemic Data Dissemination.
  Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira and Étienne Rivière.
  In IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2012.

This paper presents Brisa and the main design decisions behind the algorithm. The major result is the recognition that it is indeed possible to combine the robustness of epidemic dissemination with the efficiency of a structured approach, such as trees and directed acyclic graphs in a single algorithm. The evaluation was done in a real environment with a real implementation by leveraging the Splay framework (Leonini et al. 2009) and is publicly available at `http://www.splay-project.org/splay/ipdps2012/brisa.zip`.

- Scaling up publish/subscribe overlays using interest correlation for link sharing.
  Miguel Matos, Pascal Felber, Rui Oliveira, José Pereira and Etienne Rivière.
  In IEEE Transactions on Parallel and Distributed Systems (TPDS), 2013.

  This paper extends and improves the work done on StaN by presenting the algorithm in much more detail. StaN is evaluated with real workloads from traces of LiveJournal and Wikipedia showing substantial improvements on topic management overhead even under dynamic environments. Moreover, this paper also proposes a simple event dissemination algorithm that takes advantage of event crossposting to improve bandwidth usage. Besides the evaluation on a simulated environment, we also evaluate StaN in a real environment with the help of the Splay framework (Leonini et al. 2009). The implementation is publicly available at `XYZ`.

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

  This paper extends and improves the work done on Brisa by providing a more detailed description of the algorithm and its main properties. The focus was on the support of multiple dissemination structures, their impact on the algorithm's behavior and an evaluation comparing the different alternatives. The implementation, done with the help of the Splay framework (Leonini et al. 2009), is publicly available at `XYZ`.

- An Epidemic Total Order Algorithm for Large-Scale Distributed Systems.

Miguel Matos, Pascal Felber, Rui Oliveira, and José Pereira.
(submitted)

This paper presents the main idea leading to EpTO and a detailed evaluation of the algorithm under different scenarios and configurations on a simulated environment.

- LayStream: A Layered Approach to Gossip-based Live Streaming.
  Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, and Etienne Rivière.
  (submitted)

  This paper builds partially on the work done on Brisa and its efficiency and robustness properties to build an epidemic live video streaming system. The stringent requirements of this setting led us to implement and evaluate several key components of a typical epidemic system, such as the peer sampling and topology construction services, identify mismatches between simulated and real deployments and propose solutions for them.

Besides, during the course of this dissertation several collaborations sprung from the work done here. The most related to this disseration, in chronological order, are:

- An epidemic approach to dependable key-value substrates.
  Miguel Matos, Ricardo Vilaca, José Pereira, and Rui Oliveira.
  In International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments - Dependable Systems and Networks Workshops, (DSN-W), 2011.

- Slead: low-memory, steady distributed systems slicing.
  Francisco Maia, Miguel Matos, Étienne Rivière and Rui Oliveira.
  In International Conference on Distributed Applications and Interoperable Systems, (DAIS) 2012.

- Slicing as a distributed systems primitive.
  Francisco Maia, Miguel Matos, Etienne Rivière and Rui Oliveira.
  In Latin-American Symposium on Dependable Computing (LADC), 2013.

- DataFlasks: an epidemic dependable key-value substrate.
  Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira and Etienne Rivière.
  In International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments - Dependable Systems and Networks Workshops, (DSN-W), 2013.

1.4 Dissertation outline

This dissertation is organized around the set of papers published and submitted above. To this end, we organized the dissertation in three major chapters, one for each of the major issues addressed, namely: message types, message size and latency and message ordering. Because each theme is roughly self-contained, instead of discussing the related work in a separate chapter, we present it in the respective theme's chapter. Still, common concepts, assumptions and models are presented together in a background chapter.

In detail, the remainder of this dissertation is organized as follows:

- Chapter 2 provides a brief introduction to the fundamental concepts of epidemic systems, presents the building blocks used for this dissertation and discusses the general model and assumptions made.
- Chapter 3 presents the design and evaluation of StaN, addressing the problem of message types
- Chapter 4 presents the design and evaluation of Brisa, addressing the problem of message size and latency.
- Chapter 5 presents the design and evaluation of EpTO, addressing the problem of message ordering.
- Chapter 6 concludes the dissertation, summarizes the major contributions and gives some pointers for possible paths of future research.

Chapter 2

Background

In this chapter we introduce the concepts underlying epidemic algorithms, define our programing model and assumptions and describe the conventions used throughout the rest of this dissertation.

2.1 Model

In general, distributed systems are classified according to the interprocess communication mechanism, the timing model and the failure model (Lynch 1996).

The failure model specifies the type of faults in the system and it encompasses failure detectability, process recovery, omissions in sending and/or receiving messages and arbitrary failures known as Byzantine. For simplicity, we assume only failstop failures in which processes are allowed to fail and such failures can be eventually detected. This means that if a process fails and later recovers and rejoins the system, for instance as a consequence of churn, it does so as a new process. Moreover, in the algorithms we propose, the precision of failure detection only impacts performance not correctness. Processes that do not fail are said to be correct. Interprocess communication is the mechanism used by processes to exchange operations and data, and can be implemented by shared memory, point-to-point or broadcast of messages, or remote procedure calls. Due to its simplicity and wide availability, we focus only on point-to-point communication over an IP network using a transport protocol, such as TCP or UDP. By assumption, each process can be uniquely identified and reached by its IP address and port, i.e. we do not consider processes behind firewalls or NATs. Note that this

limitation can be overcome by the use of several techniques, using epidemic algorithms readily available (Kermarrec et al. 2009; Leitão et al. 2010). The virtual link between any two pair of processes is called a channel. We assume channels to be fair-lossy, i.e. losses might occur but if a correct process sends a message infinitely often to another correct process, the latter will receive that message infinitely many times. Moreover, we further assume that channels do not corrupt, duplicate or create spurious messages. In practice, this can be implemented by a reliable transport protocol, such as TCP, or by the application by using message retransmissions or the stubborn channel abstraction (Guerraoui et al. 1998).

The timing model is concerned with the assumptions done regarding relative process speeds and communication channels timeliness. In one extreme, we have synchronous systems which have a well-known upper bound on the time it takes for processes to execute operations and on the time taken since a message is sent until it is received. On the other hand, in asynchronous systems there is no such upper-bound and both processes execution speed and message transmission time can take infinitely long. Despite being harder to reason about, asynchronous systems are more generic and therefore we focus only on asynchronous systems in this dissertation. Moreover, unless otherwise stated, there is no assumption in the availability of a global clock.

2.2 Overlay Networks

In order for the system to function properly, each process needs to know the identifier of other processes with which it can communicate with. This set of process identifiers is known as the view and the size of the view is known as the degree. When a process p has a process q in its view, q is said to be a neighbor of p and the set of all processes in p's view is called the neighborhood of p. The set of all views establishes a *who knows who* relationship and is known as an overlay network - a logical network imposed on top of the physical infrastructure. When analyzing the global properties of an overlay network, it is often useful to model it as a graph where processes are vertices connected by the links or edges induced by the views. This graph should have some key properties that any algorithm should strive to obtain and preserve. These properties are (Jelasity et al. 2007a):

- **Connectivity:** indicates process reachability and is obtained when any

process is able to reach every other process in the system in a finite number of hops. Failure to ensure connectivity, known as a partition, severely impairs the usefulness of algorithms as not all processes are able to receive the desired application data.

- **Average path length:** measures the average number of hops separating any two processes in the system. It is related to the overlay diameter which is given by the greatest path length between any two processes. The average path length should be as small as possible as it imposes a lower bound on the time needed to disseminate data among all processes.
- **Clustering coefficient:** measures the closeness of neighbor relations among processes. It is defined as the number of links among the neighbors of a given process divided by the number of all possible links among those processes. This property affects redundancy because the number of duplicates received directly increases with it, and also robustness because graphs with high clustering coefficients are more prone to partitions.
- **Degree distribution:** is the distribution of the number of neighbors of each process - the degree or size of the view - and measures processes' reachability and their contribution to the connectivity.

In the following, we present the two main approaches to build overlay networks: structured and unstructured.

2.2.1 Structured Overlays

In the class of structured overlay networks, the neighboring relations among processes are established judiciously according to some criteria, such as latency or distance. Due to the tight control over link establishment, structured overlay networks are efficient in routing data and/or requests to the appropriate process, as the location of those processes could be calculated in a deterministic fashion. Thus, structured overlay networks are popular to store and retrieve arbitrary data and build distributed hash tables (DHT) (Plaxton et al. 1997). DHT algorithms define a topology by assigning identifiers to each process, and a function that determines the distance, in number of hops, between any two identifiers in the space. Nonetheless, the inherent overlay structure can also be used to provide

data dissemination primitives to applications (Jannotti et al. 2000; Ratnasamy et al. 2001; Zhuang et al. 2001; Castro et al. 2002). Structured overlay networks are typically built as spanning trees (Gallager et al. 1983) or more complex structures, such as hypercubes (Rowstron and Druschel 2001; Zhao et al. 2001; Stoica et al. 2003) and Cartesian hyperspaces (Ratnasamy et al. 2001).

Despite the frugality in resource consumption of both processes and links, structured overlay networks are highly sensitive to churn and failures. The frugality comes from the before hand construction of the network structure that is able to take advantage of links and processes with higher capacity. However, upon failures the overlay must be rebuilt, precluding the dissemination of data to all processes while this process takes place. As such, in highly dynamic environments where the churn rate is considerable, the cost of constantly rebuilding the overlay may become unbearable. Furthermore, processes closer to the root of the spanning tree handle most of the load of the dissemination, thus impairing the scalability of the approach. This also applies to the aforementioned structures, as certain processes become critical in reaching a large part of the network, and therefore are responsible for handling the network load of large portions of the system.

2.2.2 Unstructured Overlays

In the unstructured approach, links are established randomly among processes without taking into account any efficiency criteria. Therefore, to guarantee that all processes are reachable, and thus the connectivity property ensured, links need to be established with enough redundancy, which has a significant impact on the overlay. The main advantage is that because there are multiple paths available between any two pair of processes, failures and churn do not impair the successful delivery of a given message as it will be routed by some other available path. Furthermore, as there is no implicit structure on the overlay, the churn effect is mitigated as there is no need for global coordination or rebuilding of the overlay. These characteristics yield strong desirable properties in distributed systems: reliability, as connectivity is preserved despite faults; and resilience, as the effect of churn is negligible when compared to structured approaches. Scalability is obtained by requiring each process to know only a small subset of neighbors, typically bounded by the logarithm of the size of system, thus minimizing the

load imposed on the maintenance of the overlay and in the dissemination of application data. However, departing from global knowledge to only a partial view of the system has a serious impact on the algorithms as they need to address several design questions in order to be successful, which include uniformity and adaptivity (Eugster et al. 2004). The reliability of the overlay stems from the fact that links are established randomly among all the processes in the system. However, when the algorithm is restricted to knowledge of only a subset of processes, this uniform randomness can only be preserved if the partial view of the system is itself a uniform sample from the system. Adaptivity is concerned with the size of the partial view of the system. If the system size is known before hand then the appropriate view size can be easily determined (Kermarrec et al. 2001). However, when the system size is unknown and/or it varies along the time, the partial view size maintained by each process needs to be adapted in order to ensure that the connectivity of the overlay is preserved. Finally, the degree distribution of the overlay should be even, i.e. the variance of the average degree should be small. This is fundamental to ensure load balancing as the load imposed on processes - both in management overhead and in the dissemination effort - is closely related to the degree.

The mechanism used to construct the overlay in the unstructured approach is known as the Peer Sampling Service (PSS) (Jelasity et al. 2007b). Due to its importance as the most fundamental building block in unstructured overlays, there is an extensive body of research on building a PSS in a fully decentralized fashion (Lin and Marzullo 1999; Ganesh et al. 2001, 2002; Massoulié et al. 2003; Voulgaris et al. 2005a,c; Leitão et al. 2007; Melamed and Keidar 2008). Existing PSS proposals can be roughly classified as reactive or proactive according to the way they update the processes' view. In the proactive case, processes periodically exchange their views with their neighbors regardless of the actual need to replace failed entries, resulting in each view being a continuous stream of process samples from the network. Examples of proactive PSSs include Cyclon (Voulgaris et al. 2005a) and Newscast (Voulgaris et al. 2005c). In the reactive case, the view is kept unchanged unless some of its entries need to be updated, i.e. for replacing a failed process or for accommodating a process joining the system. Typical examples include Scamp (Ganesh et al. 2001), Araneola (Melamed and Keidar 2008) and HyParView (Leitão et al. 2007b). The trade-off between reac-

tive versus proactive strategies is essentially one between the frugality in terms of bandwidth consumption of reactive approaches versus the view freshness and diversity provided by the proactive approaches.

Random walks Many distributed algorithms over unstructured overlays often need to sample the network to collect some application specific information. This procedure can be modeled as a random walk, a graph traversal procedure (Gkantsidis et al. 2006; Massoulié et al. 2006). Briefly, a process initiates a random walk by randomly selecting a neighbor from its view and sending it a specific message. The receiver executes some application specific logic, adds some information to the one already carried in the message from the random walk, and forwards the random walk to a randomly selected neighbor. Each random walk is configured with a maximum number of hops it needs to take after which it returns to the initiator. Upon receiving the random walk, the initiator uses the information collected in an application specific manner.

2.2.3 Discussion

The trade-off between the structured approach and the unstructured one is clear. In the structured approach it is possible to take advantage of processes and links with high capabilities thus improving the efficiency of the solution. However, those approaches are sensitive to faults and churns and thus require a stable environment in order to operate properly. On the other hand, unstructured approaches are able to operate under considerable amounts of faults and churn, but the toll to pay is increased overhead when disseminating application data. The trade-off here is between a very efficient, brittle approach or a robust, less efficient one.

Because we target very large scale systems where churn is the norm rather than the exception, our design philosophy throughout this dissertation is to start with a robust unstructured algorithm and then judiciously optimize it for performance. To this end, all algorithms developed assume the existence of a PSS implemented by one of the aforementioned proposals.

2.3 Data Dissemination

The goal of constructing an overlay network, regardless of the particular approach taken, is usually to offer its capabilities to other services able to disseminate application data from one or more sources. In this section we briefly introduce different data dissemination algorithms and highlight the trade-offs among them. We consider three different approaches, namely flooding, trees and epidemic algorithms. Because these approaches rely on the membership information provided by the overlay network, there are naturally some combinations more adequate than others while others overlap in terms of functionality. For instance, flooding a structured overlay with the shape of a tree is similar to using a tree dissemination strategy on an unstructured overlay network. Nonetheless, because these approaches are at different abstraction levels, we conceptually separate them.

2.3.1 Flooding

Flooding is the simplest dissemination strategy. Essentially, all application messages received are relayed to all neighbors on the overlay network. As expected, flooding is very demanding in bandwidth and as such, several optimizations to this naive strategy exist that take advantage of the location of processes in order to reduce the number of duplicates received. In one of those strategies, flooding is only done in the same 'direction' as the received message, as processes on the opposite direction are already expected to have received the message (Ratnasamy et al. 2001).

2.3.2 Tree

In tree approaches, such as (Castro et al. 2002), the dissemination of application level messages uses a reverse path forwarding mechanism to construct and maintain the multicast group, encompassing all processes interested in the dissemination. For each multicast group, the dissemination protocol creates a multicast tree with a unique identifier and uses it to relay messages to the relevant processes. To join the group, a process uses the overlay network to send a message to the multicast group. As the joining request traverses the overlay, each process checks whether it is already part of the desired multicast group, and if it is, it

stops forwarding the message and adds the joining process as a child in the multicast tree. If not, the request is forwarded to the parent until it is adopted by a process or it reaches the root of the tree. In the latter, the root will adopt the joining process as a direct child. The protocol carefully balances the multicast tree in order to ensure an evenly load distribution among the participating processes. To further prevent bottlenecks in certain processes, the protocol provides mechanisms to demote a process's child to a grandchild, thus transferring some of the dissemination effort to its children. Further details of the deployment of these protocols on top of the structured overlay construction mechanisms available, and a detailed comparison of the trade-offs between each one can be found in (Castro et al. 2003c). As the mechanism used to construct the dissemination tree ensures loop-free paths, there are no message duplicates delivered to the application.

2.3.3 Epidemic

Epidemic or gossip dissemination approaches rely on the mathematical models of epidemics (Bailey 1975; Demers et al. 1987; Birman et al. 1999; Eugster et al. 2003b, 2004): if each infected element spreads its infection to a number of random elements in the universe, then all the population will be infected w.h.p. The number of elements that need to be infected by a given element is called the fanout and is a fundamental parameter of the model. Note that even if the model specifies that the elements to be infected need to be selected uniformly at random from the universe, processes usually know only a small fraction of all processes - those in their view. This is addressed by works such as lpbcast which ensure that the view of processes has the same properties than a uniform sample of all processes (Eugster et al. 2003b). Thus, processes pick $fanout$ elements from its view and send the message to them. The choice of the value of the fanout highly influences the fraction of the population that becomes infected. As specified in (Eugster et al. 2004), the ideal fanout value defines a phase transition: below that value the dissemination will reach almost no processes, and above it the dissemination will reach almost all processes. The decision of *when* and *how to* send the message payload to the chosen processes may follow several approaches (Karp et al. 2000), which we describe next. In the *how* to send the message decision there are two options available: push and pull. With push the sender takes the initiative and relays the message to its neighbors as soon as it is

received. On the other hand, with pull the receivers ask periodically the sender for new messages, which will then relay any new message to the receiver. In the *when* decision there are also two options: eager and lazy. Essentially this defines if the message payload should be sent immediately, the eager variant, or only an advertisement of the message, the lazy variant. When combining both design decisions we have four options:

- **Eager push:** the message payload is sent as soon as it is received. This minimizes latency, but at the expense of bandwidth as processes are likely to receive many duplicates. It is the most common strategy and is used by several well-known protocols, such as (Ganesh et al. 2001; Eugster et al. 2003b; Pereira et al. 2003).

- **Lazy push:** upon reception of the message payload the process sends an advertisement of the message to its neighbors. Interested processes can then ask the sender for the payload. In this approach the latency increases considerably as three communication steps are necessary to receive the payload, in a pure lazy push system duplicates are eliminated. This strategy is used in protocols, such as (Liu and Zhou 2006; Carvalho et al. 2007).

- **Eager pull:** periodically processes will ask their neighbors for new messages. Upon reception of the request, processes will send all new messages to the requester. As in the push variant, this approach minimizes latency but at the cost of high bandwidth usage. It is used in protocols, such as (Nguyen et al. 2010; Frey et al. 2010).

- **Lazy pull:** periodically processes will ask their neighbors for new messages. Upon reception of the request, processes will send a message with the identifiers of all new known messages to the requester, who can then selectively pull the relevant messages. This strategy is also known as two-phase pull and allows for an optimal use of bandwidth even though its latency is considerable. It is used in the Network News Transfer Protocol (Feather 2006), which powers Usenet.

The eager versus lazy strategy is clearly a trade-off between bandwidth and latency, while the difference between a push and pull scheme is more subtle. With push processes behave reactively to message exchanges, while with pull

processes behave in a proactive fashion by periodically asking for new messages. Thus, in an environment where messages are sparing, a push strategy has no communication overhead, while the pull approach presents a constant noise due to the periodically check for new messages. Proposals such as (Pianese et al. 2007; Carvalho et al. 2007; Wang et al. 2010) try to overcome the disadvantages of each strategy by combining them in the same protocol.

2.3.4 Discussion

Tree approaches are very efficient in bandwidth usage as, by construction, they avoid sending and receiving message duplicates. Furthermore, by manipulating the depth and branching factor of the tree it is possible to obtain a wide range in end-to-end latency at the cost of putting more load on the interior processes of the tree. However, similarly to structured overlay networks, trees are vulnerable to faults and churn, as the failure of an interior process will preclude the reception of messages in its entire sub-tree. On the other hand, the flooding approach is completely oblivious to faults and churn, as long as the overlay network is connected, all processes will receive all messages. The cost of this resilience is however a large amount of duplicates received, as each process will receive as many copies of a given message as the view size - one for each neighbor. Technically, in the tree there is also a flooding process through its branches, however this is done only to the selected processes (the ones that define the tree according to the propagation strategy), whereas in a pure flooding the message is sent to all the neighbors obtained from the overlay network. Epidemic approaches present an interesting mid-term between the two extremes. The resilience is comparable to flooding, however, they are much less demanding in terms of bandwidth usage. With the use of proper strategies, epidemic approaches can even offer a bandwidth usage similar to the tree, where no duplicates are received.

2.4 Conventions

For readability, we use some conventions throughout this dissertation mostly regarding presentation style.

When presenting algorithm listings we use the following keyword conventions:

- **initially:** invoked when the process starts, used to initialize data structures
- **every δ:** invoked every δ time units, usually contains the main loop of the algorithm
- **procedure:** invoked locally by the process
- **send MSG to p:** sending of a message MSG from the current process to target process p
- **upon receive MSG:** invoked when a message MSG is received by the current process
- **RandomPick(lst):** picks an element uniformly at random from the list *lst*.

In the literature one can often find the terms peer, process, processor, node or machine to refer to slightly different concepts. Technically, a node, machine or processor is the physical hardware. On top of that we have processes or peers (software) participating in a given distributed algorithm. While it is possible to have several processes running on the same node, for simplicity we do not make such distinction and use all the terms interchangeably.

Finally, one can also find in the literature the related terms message and event. In this dissertation, we consider an event to be a piece of information created and delivered by a process, while the message is the network level entity (usually an Ethernet frame) carrying one or more events.

Chapter 3

StaN: scalable topic-based publish-subscribe

3.1 Introduction

As society becomes ever more digital, the number of users connected to the Internet increases and the variety of data generated online grows steadily. Consequently, there is a huge demand in dissemination systems responsible for delivering data to their intended recipients in a variety of contexts, ranging from social networks and news sites to enterprise environments and financial markets.

The publish/subscribe paradigm emerged as an attractive model for scalable event dissemination, mainly due to the strong decoupling between the communicating entities: the producers (publishers) and consumers (subscribers) of information (Eugster et al. 2003a). This flexibility—in terms of space, time and synchronization—makes this model suitable to a wide range of application domains, from collaborative feed dissemination systems (Jun and Ahamad 2006; Nunes et al. 2009) to enterprise service bus middleware used in service-oriented architectures (Barazzutti et al. 2013).

The topic-based publish-subscribe variant categorizes items by explicit topics, avoiding the overhead of content-based filtering (Voulgaris et al. 2006). Topics act as named channels where content can be published in the form of events. Participants interested in specific content subscribe to one or more topics and subsequently receive all the events published in these topics. Albeit simple, it captures the behavior of many real world scenarios, such as Usenet, Web syndi-

cation, Wikipedia, and social networks, such as LiveJournal.

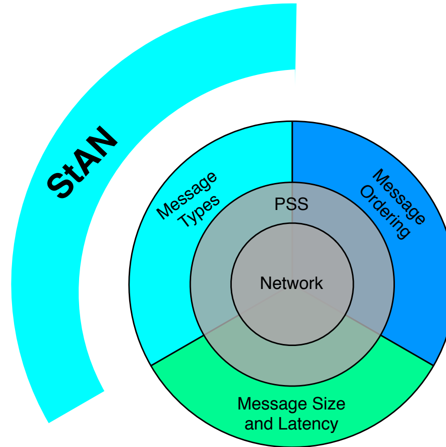


Figure 3.1: StaN placement in the problem space.

Topic-based publish/subscribe has attracted much interest among researchers (Chockler et al. 2007a,b; Baldoni et al. 2007a; Patel et al. 2009a), especially on the design of decentralized approaches that do not rely on a central broker to manage topics and route events. Furthermore, due to scale and dynamic behavior, many existing proposals rely on epidemic dissemination (Birman et al. 1999; Eugster et al. 2003b) to implement topic-based publish/subscribe communication (Eugster et al. 2003a).

Some designs build several stacked overlay networks, one for each topic, and have processes independently join overlays for each of its subscriptions using traditional epidemic protocols (Birman et al. 1999; Eugster et al. 2003b; Jelasity et al. 2007c). Unfortunately, this has high maintenance costs and presents scalability problems as the number of links established by each process grows linearly with the number of subscribed topics. Moreover, publishing the same event in multiple topics yields redundant transmissions as events are separately disseminated among the same processes through different overlays.

The alternative is to maintain a single overlay so that processes with similar interests are close to each other (Chand and Felber 2005; Chockler et al. 2007b). Shared interests are explicitly taken into account and redundant event transmissions on multiple topics avoided. With global knowledge of process interests, such semantic clustering can be achieved using epidemic interactions (Massoulié et al. 2003; Jelasity et al. 2009). This has been formalized as the *minimum topic-connected* problem (Chockler et al. 2007a) and shown to be NP-complete. Furthermore, the resulting overlay is likely to exhibit a high clustering coefficient due to the approximation of processes with similar interests. Therefore, it will be highly sensitive to faults and churn, and prone to partitioning (Jelasity et al. 2007c).

In this chapter we present StaN, a novel approach to topic-based publish-subscribe that aligns multiple independent overlays in order to promote link sharing among them. Despite being managed independently and in a decentralized fashion, provided that there is subscription correlation, the overlays converge to share a large number of links. The growth is slower with the number of topics than traditional multi-overlay approaches, thus promoting topic scalability. This is achieved while preserving the desirable properties for epidemic dissemination, namely low clustering coefficient and low diameter thus making StaN an attractive infrastructure for efficient and scalable topic-based publish-subscribe.

The rest of this chapter is organized as follows. We present StaN and a dissemination algorithm leveraging link sharing in Section 3.2 and evaluate both algorithms in Section 3.3. Related work is discussed in Section 3.4 and finally Section 3.5 concludes the chapter.

3.2 Algorithm description

In this section, we describe the system model, present StaN and discuss its main properties.

3.2.1 System Model and Assumptions

To map as close as possible to real-world observations, we assume that topic popularity (the number of subscribers for a topic) and subscriptions per process (the number of topics a process is subscribed to) follow power law distributions (Liu

et al. 2005). We also assume that topic subscriptions are correlated, i.e. there is a non-negligible probability that subscription sets overlap as observed in real workloads (Saroiu et al. 2002; Fraigniaud et al. 2005; Handurukande et al. 2006). We note that in the absence of correlation, StaN will degenerate in an overlay-per-topic solution. In this case even single overlay approaches will produce disconnected components (one per topic). StaN's performance is therefore ultimately driven by the number of subscriptions per process and the correlation between topics.

StaN's architecture is presented in Figure 3.2. Our approach assumes each topic has a separate random (unstructured) overlay network, maintained by some peer sampling service (PSS) like Scamp (Ganesh et al. 2001). The key properties of these overlays are: *i)* average view size grows logarithmically with the system size, thus enabling *process scalability*, and *ii)* clustering and diameter are low, making the overlays *fit* for epidemic event dissemination and resilient in face of faults and churn (Jelasity et al. 2007b). Choosing processes uniformly at random is key to ensure those properties (Eugster et al. 2004) and, therefore, it is fundamental to preserve randomness when combining links from different overlays. Link combination and alignment is done by the *Link Management* component described in Section 3.2.3.

We model an overlay as a directed graph and assume the PSS maintains the graph connected. Using directed graphs allows each participant to make strictly local decisions regarding the management of its *view* by establishing and removing links as appropriate.

Overlay links are a *logical* abstraction of the underlying network, which are mapped to a *physical link* by transport protocols, such as TCP or UDP. This can be implemented by a dynamic pool of shared TCP/IP connections as in NeEM (Pereira et al. 2003). Therefore, an epidemic dissemination protocol leveraging StaN can exploit logical links on different overlays that share the same physical link, thus avoiding redundant retransmission of the same event. This is achieved by the *Dissemination Management* component described in Section 3.2.4.

For simplicity we do not differentiate publishers from subscribers and assume both are interested in receiving all events published on the topic. With this model of all-to-all communication we use process as a means for both a publisher and a

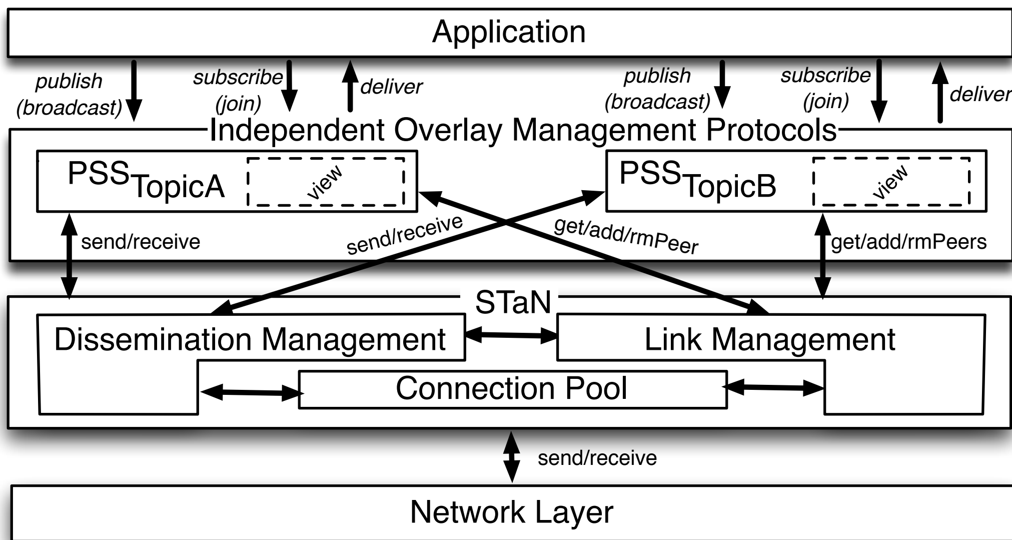


Figure 3.2: StaN's architecture.

subscriber.

3.2.2 Design Rationale

When designing a topic-based publish/subscribe system, a set of desirable properties naturally emerge:

- **Completeness:** A process should receive all events published in the subscribed topics.
- **Accuracy:** A process should not receive any event from a topic it is not subscribed to.
- **Process scalability:** The system should scale with respect to the number of processes for a given topic.
- **Topic scalability:** The system should scale with respect to the number of topics.
- **Fitness:** The overlays should have good structural properties to enable efficient event dissemination and resilience to faults and churn. As discussed in Section 2.2, these properties are: *connectivity*, *average path length*, *degree*

distribution and *clustering coefficient*. *Connectivity* indicates that all processes are reachable from any other process and is fundamental to ensure completeness. *Clustering coefficient* is related to the dissemination cost as highly clustered portions of the overlay will produce more redundant messages, and to fault tolerance as highly clustered sections of the overlay tend to easily become disconnected from each other, thus compromising overall connectivity. *Average Path Length* gives a lower bound on the time and cost for a message to reach all processes. *Degree distribution* measures the number of neighbors of a process and thus its reachability and load.

Proposals based on a single overlay, such as SpiderCast (Chockler et al. 2007b), recognize and exploit common subscriptions among processes, allowing the number of links to grow sub-linearly with the number of topics and processes thus providing *process scalability* and *topic scalability*. However, the remaining properties are more challenging to maintain. In particular, the *fitness* of the overlay degrades because semantic communities lead to high clustering coefficients. *Accuracy* is also problematic as, eventually, processes will have to relay events they are not interested in to guarantee *completeness*.

On the other hand, proposals like daMulticast (Baehni et al. 2004) or TERA (Baldoni et al. 2007a) that build one overlay per topic satisfy *completeness* and *accuracy* as each event is disseminated completely and only through the overlay it belongs to. *Fitness* and *process scalability* are also satisfied as these approaches rely on epidemic PSSs designed for scalability and epidemic dissemination (Eugster et al. 2003b; Jelasity et al. 2007b). The major drawback is *topic scalability* as the number of physical links established grows linearly with the number of topics each process subscribes to. Moreover, if an event matches more than one topic, these approaches cannot exploit this knowledge to reduce traffic because topics are fully separated.

StaN seeks to achieve the best of both worlds by addressing all aforementioned properties by combining several logical links in a single physical link. These combinations, detailed in Section 3.2.3, are strictly local decisions made by each process based on the set of other processes in a topic-overlay. Link combination also allows an event published on multiple topics to be relayed just once through the *physical link* as detailed in Section 3.2.4.

3.2.3 Link Management

The intuition behind StaN is very simple: each process periodically samples the subscribers of all its topics, that is, at each overlay it belongs to. Since, by assumption, subscriptions are correlated these sets of sampled processes will likely overlap. For each overlay, the process then deterministically selects a set of neighbors from the sampled processes. This deterministic selection over overlapping sets leads, with high probability, to neighbors shared across all overlays enabling the mapping of several logical links to a single physical link, thus alleviating topic scalability problems.

However, such design raises two conflicting goals: first we want to promote link sharing by taking advantage of subscription correlation and second we do not want to induce clustering (due to subscription correlation) as this will impact the *fitness* of the overlay. The problem at hand is then to devise a neighbor selection process able to meet both goals. In the following we study its key requirements.

To guarantee *fitness*, PSSs establish links *uniformly* at random (Ganesh et al. 2001; Eugster et al. 2003b, 2004; Jelasity et al. 2007b), and thus our neighbor selection process must preserve this randomness. Unfortunately, this implies that the probability that any two processes are logically linked in more than one overlay is dismayingly small. Even with global knowledge of the system and full disclosure of the subscription sets, finding a minimal solution (with the smallest number of physical links) is NP-complete (Chockler et al. 2007a).

On the other hand, to promote link sharing we need *determinism*, which is apparently conflicting with uniformly random choices. In fact, due to overlap in subscription sets resulting from correlation, a process using the same deterministic criterion in all overlays will independently choose approximately the same neighbors for each overlay. Besides, to avoid clustering, processes cannot choose the same set of neighbors (and neighbors of neighbors) which requires *asymmetry* in the local choices made by processes.

In our approach each process selects neighbors using a *pseudo-random* criterion that meets the above requirements. Uniformity is key to preserve the good properties of a random overlay (Eugster et al. 2004), while determinism is necessary to guarantee that a process will assign the same value to a target process independently of the overlay. Both are found in *hash functions* (Luby 1994), which produce uniform outputs along its codomain and always map the same

Algorithm 1: Pseudo-random weight function on process p for target q

```
1 procedure WEIGHT(q)
2   return HASH( STR(p) + STR(q))
```

inputs to the same outputs. Thus, by feeding a hash function with the identifiers of known processes, any process can obtain a uniform and deterministic sorting of all other processes. Still, this is not sufficient as each process must have a different sort order, otherwise the overlay would degrade into a chain-like structure. Besides, sorting needs to be asymmetric to prevent clustering among neighbors. This is obtained by having each process supply a different input to the hash function, thus yielding different sorting orders. The pseudo-code of our neighbor selection criterion (weight function) is shown in Algorithm 1. Note that Str(p) is the text representation of p's identifier.

The weight a process p assigns to a process q is given by the output of the hash function. Weights are assigned by concatenating p and q unique identifiers expressed as strings. Thus, processes can locally order all other processes and give preference to different sets of neighbors. Note finally that the weight function is asymmetric: considering any two processes p and q, $\text{Hash}(\text{Str}(p) + \text{Str}(q))$ and $\text{Hash}(\text{Str}(q) + \text{Str}(p))$ yield different values with high probability (Luby 1994).

The remaining challenge is to design a protocol that enables processes to discover neighbors with minimum weight and replace links accordingly to reach the desired configuration. The asymmetry and absence of clustering precludes the use of well-known methods, such as T-Man (Jelasity et al. 2009), that rely on the establishment of a partial order among *all* processes and dynamically converge the overlay toward a global target topology. As the weight function defines multiple orderings, one for each process in the system, there is no target topology and thus no such convergence guarantee.

Our proposal relies instead on random walks to obtain uniform samples of the population. These samples are locally ordered according to the weight function and the neighbors for each overlay chosen accordingly.

The pseudo-code for StaN is shown in Algorithm 2. Each process p accesses the list of the topics it is subscribed to through $p.topics$, and per each topic t, each process maintains a separate view, denoted by $p.views[t]$.

Algorithm 2: STAN protocol (process p)

```
   // Periodic refreshing of the views
1  every δ
2      foreach topic t ∈ p.topics do
3          q ← RandomPick(p.views[t])
4          send CollectWalk(p, ∅, TTL, t) to q

   // Random walk to collect processes
5  upon receive CollectWalk(src, set, ttl, topic)
6      set ← set ∪ {p}
7      foreach process n ∈ p.views[topic] do
8          set ← set ∪ {n}
9      if ttl > 0 then
10         q ← RandomPick(p.views[topic])
11         send CollectWalk(src, set, TTL − 1, topic) to q
12     else
13         send CollectReply(set, topic) to src

   // Reply from last process in random walk
14 upon receive CollectReply(set, topic)
15     viewSize ← |p.views[topic]|
16     list ← {q ∈ set ∪ p.views[topic] sorted using Weight(q)}
17     p.views[topic] ← first viewSize processes from list
```

The protocol proceeds as follows. Periodically, each process initiates a random walk with a given *time-to-live* (TTL) in each overlay it belongs to. It selects a random process in that topic neighborhood and sends it a CollectWalk() message with its unique identifier, an empty set that will collect other processes' identifiers, the desired TTL and the topic identifier (lines 1–4).

Upon reception of a CollectWalk() (lines 5–13), each process adds its own identifier and that of its neighbors to the received set, and forwards it to a random neighbor provided that the TTL has not expired yet. Adding the neighbors to this set improves convergence time as more identifiers are collected by each random walk. When the TTL expires, the random walk ends and the process sends a CollectReply() with the set of identifiers back to the originator process.

Upon reception of this set (lines 14–17), the process computes its view size, merges the collected set with its own view, sorts the elements according to their weight, and finally selects the best processes to replace its existing view without changing its size.

A simplified run of StaN with five processes and two topics is depicted in Figure 3.3. Process p_0 is subscribed to topics A and B and initially maintains four logical links, two for each topic, to neighbors $p_1 - p_4$ (top figure). As there is no overlap in logical links, the number of physical links is also four. When

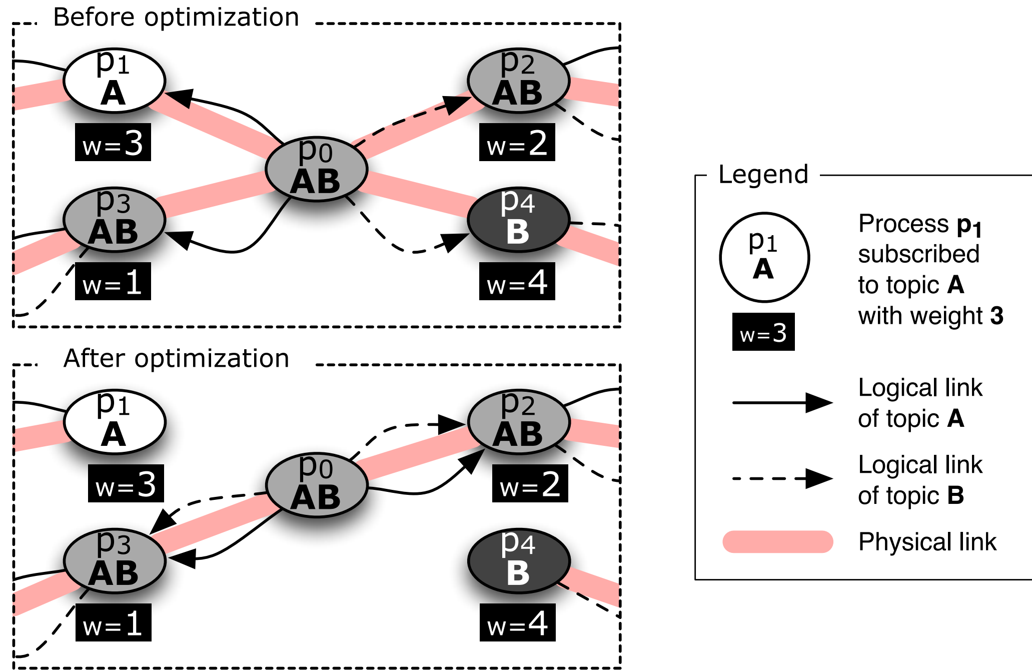


Figure 3.3: 5-processes sample run with two topics from the point of view of process n_0 (only a subset of the links is shown).

executing StaN, p_0 collects the ids of neighbors $p_1 - p_4$ and assigns then the weights shown in the figure. Then, p_0 replaces links to higher weight processes with links to lower weight ones, on a per-overlay basis. Because of correlation, the logical links of overlays A and B converge to the same physical links (bottom figure): logical links are preserved and the number of physical links is reduced. Due to asymmetry, the weight p_0 assigns to say p_2, is unrelated to the weight p_2 assigns to p_0. This is reflected in the figure by not having p_2 and p_3 choose p_0 as its neighbor, nor each other.

3.2.4 Dissemination

StaN maps several logical links to a single physical link to improve resource usage and topic scalability. This is possible due to the non-negligible probability of having overlap among processes' interests, as observed in many real scenarios (Saroiu et al. 2002; Fraigniaud et al. 2005; Handurukande et al. 2006). Besides, further studies point out that not only processes' interests overlap but

also the occurrence of messages posted to multiple topics, a phenomena known as crossposting, is non-negligible (Whittaker et al. 1998; McGlohon 2010). For instance, on Usenet at least 30% of the messages are crossposted and the average crossposted message targets 3 topics (Whittaker et al. 1998; McGlohon 2010). Consequently, a crosspost-aware dissemination protocol may be able to reduce bandwidth usage by combining crossposted messages with StaN's link sharing.

In the remaining of this section we analyze the design of such a protocol, which we call CrosspostFlood. For simplicity it is an infect and die flooding protocol, i.e. the first time it receives a message it relays it to all neighbors on the given topic(s). The only assumption is access to the list of topics a message is posted to, which can be easily included as metadata.

The basic idea is very simple and depends only on local knowledge: when the topics of a crossposted message (or a subset of it) matches a mapping of logical links to a physical one, only a single message copy is sent through the physical link. Upon reception, it suffices to deliver that message to the relevant topics.

As an example, suppose process p_0 on the bottom of Figure 3.3 receives (or creates) a message m tagged with topics A and B. Process p_0 needs to relay the message to neighbors p_2 and p_3 to topic A and do the same to topic B. Instead of sending two copies of m through each logical link to each neighbor, p_0 sends a single copy to either logical link. Upon reception of m, p_2 and p_3 detect that it has been posted to topics A and B (by observing m's metadata) and locally deliver m to topics A and B, effectively reducing the number of messages in transit from four to two. It is important to note that, although independent from the link alignment promoted by StaN, the dissemination is most effective when combined with crosspost detection. For instance, the exact same run on a non optimized version of the overlays (top of Figure 3.3) would not bring any bandwidth savings.

The pseudo-code for CrosspostFlood is shown in Algorithm 3. Each process is able to deliver an event e to a topic T by invoking *T.receive(e)* as depicted in Figure 3.2. To avoid delivery of duplicates, each process maintains a set of previously known messages, $receivedMessages$, initially empty (lines 1–2). The management of this set to avoid infinite growth is out of the scope of this paper and can be done using techniques, such as (Koldehofe 2003). A message m is generated with a unique identifier, $msgId$, and the set of topics it belongs to, $msgTopics$.

Algorithm 3: CrosspostFlood protocol (process p)

```
1  initially
       // Contains received message identifiers, to avoid duplicates
2      receivedMessages ← ∅

   // Message reception
3  upon receive Msg(msgId, msgTopics, msgData)
4      if msgId ∉ receivedMessages then
5          receivedMessages ← receivedMessages ∪ {msgId}
           // Set that will contain the processes to forward the message to
6          relayProcesses ← ∅
7          foreach topic t ∈ p.topics ∩ msgTopics do
               // Deliver message to topic t
8              t.receive(msgId, msgData)
               // Collect the processes subscribed to topic t
9              foreach process n ∈ p.views[t] do
10                 relayProcesses ← relayProcesses ∪ {n}
           // Relay the message
11         foreach process n ∈ relayProcesses do
12             send Msg(msgId, msgTopics, msgData) to n
```

Upon reception of a message (Msg), a process first checks if the message is new by observing the set of known message identifiers, and discarding it otherwise (lines 4–5). The message is then delivered to the process's topics that match the message topics, $msgTopics$ (lines 7–8). Additionally, for each matching topic t the process collects the identifiers of its neighbors in each topic in a set called $relayProcesses$ (lines 9–10).

Finally, the message is relayed to this set of neighbors as usual. By first collecting the neighbors that a message needs to be relayed to in a set, and only then sending it effectively, we eliminate possible duplicates (i.e. a process that is a neighbor in two topics), thus avoiding redundant transmissions.

3.3 Evaluation

In this section we evaluate StaN using synthetic and real workloads by simulation and via a real deployment on PlanetLab. The evaluation is focused on performance and fitness. By evaluating the performance of StaN, we are able to infer its ability to promote link sharing among overlays which is fundamental to alleviate resource consumption in the form of physical links established, and thus promote topic scalability. Moreover, we also evaluate StaN under message loss and churn to assesss its behavior on a dynamic environment and compare

it with a global-omniscient approach that aims at maximizing link sharing. The goal is to observe the extent of StaN's improvement and the inherent impact of the fitness of the overlays of such approaches. Finally, we study the behavior of the CrosspostFlood dissemination algorithm and how it takes advantage of link sharing.

3.3.1 Experimental Data

We used two real-world workloads: a trace of RSS subscriptions from LiveJournal (LiveJournal, Inc. 2013) and a trace of edits of the English version of Wikipedia (Wikimedia Foundation 2013). LiveJournal is a social network where users have a journal/blog in which they publish entries and can follow (subscribe to) the journals of others. The data gathered includes the list of users and of subscribers to the journals. This collection of users and journals, with 28,904 journals and 301,315 users, forms the complete universe of our experiments. Journals map to topics and users to processes subscribing those topics. For Wikipedia we gathered the pages and page edits done by registered users until April 2012 resulting in 715,710 pages and 2,015,060 users. Pages map to topics and users who edited those pages to processes subscribing those topics.

To increase the tractability of the universes and decrease experiment running time, we created smaller self-contained universes. A self-contained universe is created by selecting a random subset of topics, the seed set. We then select the users subscribed to topics in the seed set and add to the seed set the users' topic (journal) in LiveJournal's case or a random topic from the process's subscriptions in Wikipedia's case. The users subscribed to topics in the seed set comprise our universe pruning topics with less than 30 subscribers. As the self-contained universes were built using a random set of topics, the properties of subscription distribution are preserved. Further details on this universe generation methodology can be found in (Patel et al. 2009a). Table 3.1 describes all the LiveJournal and Wikipedia universes considered in the experiments. These workloads have been chosen as representatives of publish/subscribe systems with several seed set sizes. For each seed set size we generated 100 universes, computed the ratio between the number of topics and processes and picked the median universe. Note that $\mathcal{L}_0$ and $\mathcal{W}_0$ represents the whole LiveJournal and Wikipedia universes, respectively.

LiveJournal

Name	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$	$\mathcal{L}_5$	$\mathcal{L}_6$	$\mathcal{L}_7$	$\mathcal{L}_8$
Seeds	all	all	20,000	15,000	10,000	5,000	1,000	500	100
Topics	28,904	13,652	13,129	12,608	11,674	9,331	3,215	1,805	253
Procs.	301,315	267,230	237,612	214,642	182,828	130,577	40,407	23,657	4,689

Wikipedia

Name	$\mathcal{W}_0$	$\mathcal{W}_1$	$\mathcal{W}_2$	$\mathcal{W}_3$	$\mathcal{W}_4$	$\mathcal{W}_5$	$\mathcal{W}_6$	$\mathcal{W}_7$	$\mathcal{W}_8$
Seeds	all	100,000	50,000	20,000	10,000	5,000	1,000	100	20
Topics	715,710	328,145	245,977	162,448	114,646	77,338	26,525	3,858	576
Procs.	2,015,060	761,225	497,854	277,474	175,152	108,620	33,956	5,957	1,381

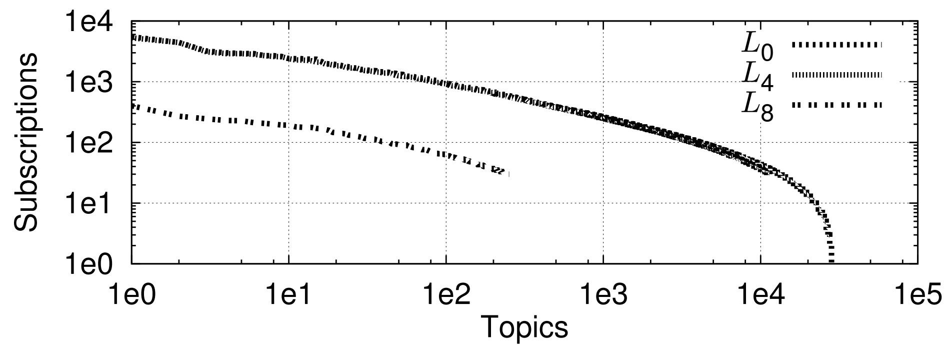
Table 3.1: Universe configurations.

3.3.2 Workload Characteristics

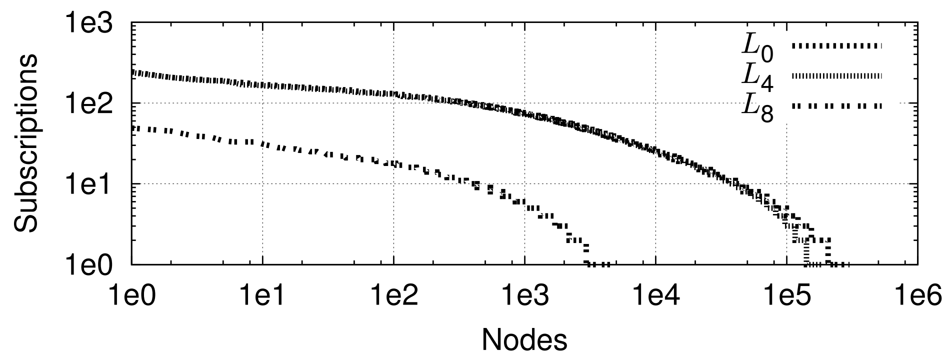
We start by confirming that our assumptions about subscriptions distribution (Adamic and Huberman 2002; Liu et al. 2005) and correlation (Saroiu et al. 2002; Fraigniaud et al. 2005; Handurukande et al. 2006) hold. These assumptions are: 1) the number of subscribers to a given topic follows a power law, 2) the number of subscriptions of each user follows a power law, and 3) subscriptions are correlated with a non-negligible probability.

Figures 3.4 and 3.5 depict the distribution of subscriptions per topic (top) and subscriptions per process (bottom) for several LiveJournal and Wikipedia universes, respectively. Note that both plots are log-log. The general shape for both LiveJournal and Wikipedia is similar: few topics are highly popular while the vast majority has few subscribers, and some users are subscribed to many topics while most subscribe to far fewer topics. These results confirm our assumptions about subscription distribution (Adamic and Huberman 2002; Liu et al. 2005) and validate our method for the generation of smaller universes. Users subscribed to many topics are the ones that can encounter problems with topic scalability, as the number of physical connections they need to maintain can be quite large. STAN is therefore expected to mainly affect these processes.

Figure 3.6 depicts the correlation among subscriptions as a heat map, where white means no correlation and black strong correlation. It was obtained by creating a matrix with topics as columns and subscribers as rows. For each

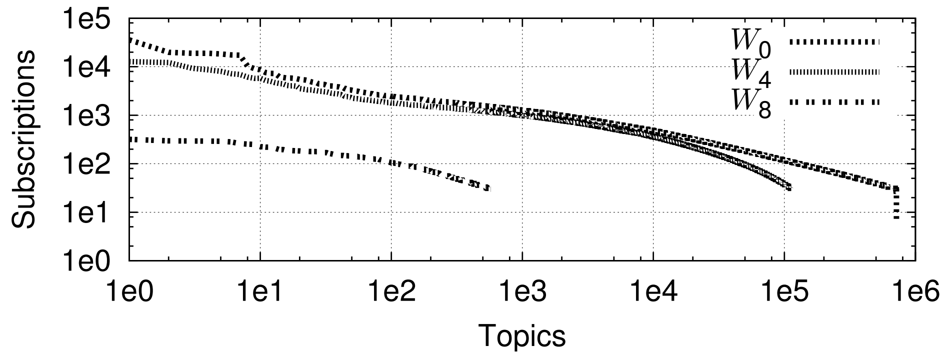


(a) Subscriptions per topic.

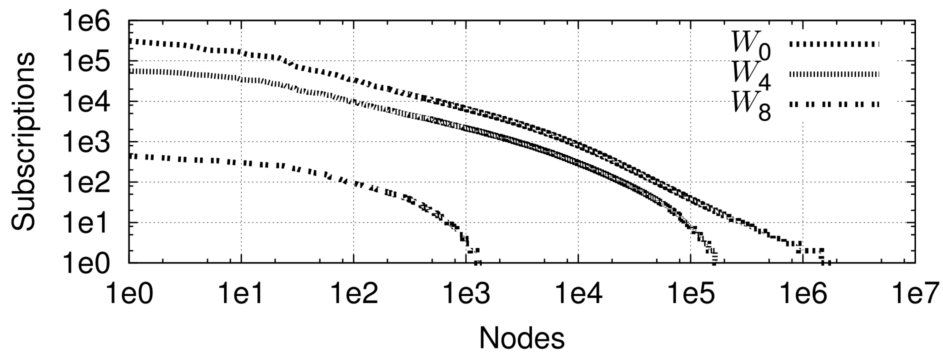


(b) Subscriptions per process.

Figure 3.4: Subscription distribution for LiveJournal universes.



(a) Subscriptions per topic.



(b) Subscriptions per process.

Figure 3.5: Subscription distribution for Wikipedia universes.

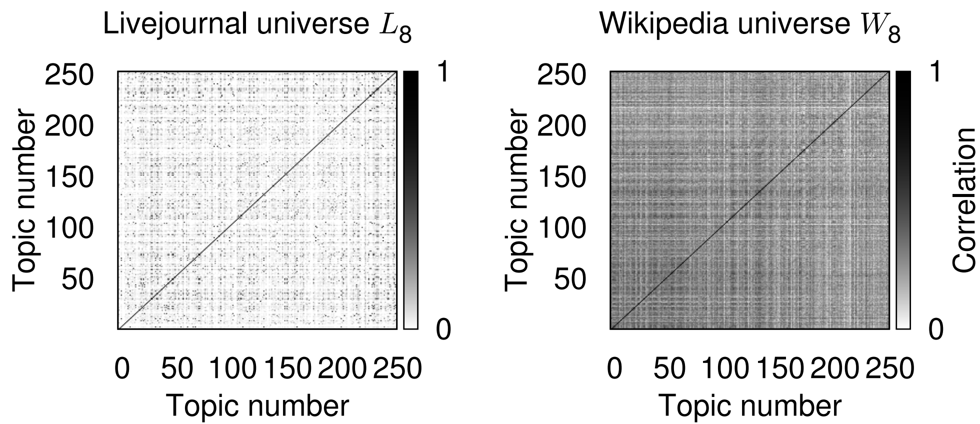


Figure 3.6: Subscription correlation.

subscriber, we set the value 1 in the respective column to indicate a subscription to the given topic, or 0 otherwise. We then calculate the Pearson correlation of the resulting matrix and plot it by mapping the values to different shades of gray. For the LiveJournal universe (top) there is a mild correlation among all topics (the map contains a non-negligible amount of gray points) while for the Wikipedia universe (bottom) the correlation is stronger. This indicates that StaN should be able to promote physical link sharing on both universes, but to a greater extent on Wikipedia.

Finally, we devised a synthetic workload that provides finer control on the number of topics/processes in a given universe to cope with the limitations of the PlanetLab (PlanetLab 2013) testbed. To this end, we built a two-dimensional grid with randomly placed process and topic identifiers. Additionally, each process is assigned an interest radius, and subscribes to topics whose identifiers fall within. For each topic, we randomly place several topic identifiers on the grid. The number of topic identifiers follows a power-law, thus matching the topic popularity model. The assigned interest radius also follows a power-law, thus matching the process subscription model. Nodes close on the grid are likely to subscribe to the same topics, hence modeling subscription correlation. This synthetic workload closely matches our model and exhibits distributions similar to those observed on real universes (Matos et al. 2010).

3.3.3 Experimental Setup

We evaluate StaN both through a real deployment on PlanetLab and by simulation.

The deployment on PlanetLab is done with Splay (Leonini et al. 2009), a framework for the development, deployment and evaluation of distributed applications. Splay handles all the details of setting up the testbed, deploying and running the job and collecting the metrics, thus greatly facilitating the prototyping and testing of distributed application and algorithms. The code is written in the Lua programing language and remains close to the pseudo-code presented in the algorithm listings. Moreover, Splay has a churn module that enables the reproduction of experiments subject to churn.

However, due to the scalability and resource limitations of PlanetLab it is not possible to perform all experiments on a real deployment. To overcome this

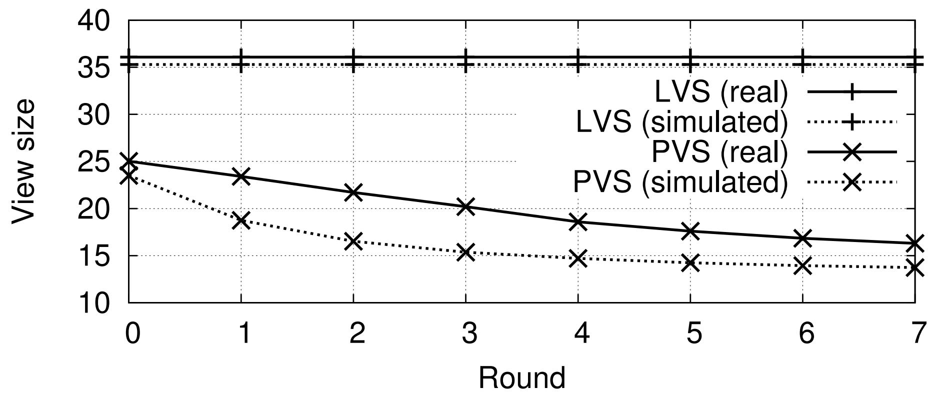
limitation we developed a realistic discrete, round-based event simulator, called POS, that can scale to thousands of processes, and thus simulate the large scales we aim at. POS models network asynchrony, process drift, and churn and can be found at `TODO`. It uses a priority queue and a monotonically increasing integer to represent the passage of time, a tick.

The Splay code used throughout the evaluation can be found at `TODO` and the POS code at `TODO`. To assess the accuracy of the POS simulator, we first conducted a series of experiments with the same workload on both Splay and the POS. For a given workload, we first created the overlays for each topic by having every process, either real or simulated, choose *viewSize* neighbors randomly. *viewSize* was configured such that the probability of the overlay being connected is 0.99 as specified in (Kermarrec et al. 2001). Finally, we ran StaN for eight rounds and collected results. We used rounds of 30 seconds for Splay, corresponding to a discrete time step of the simulator.

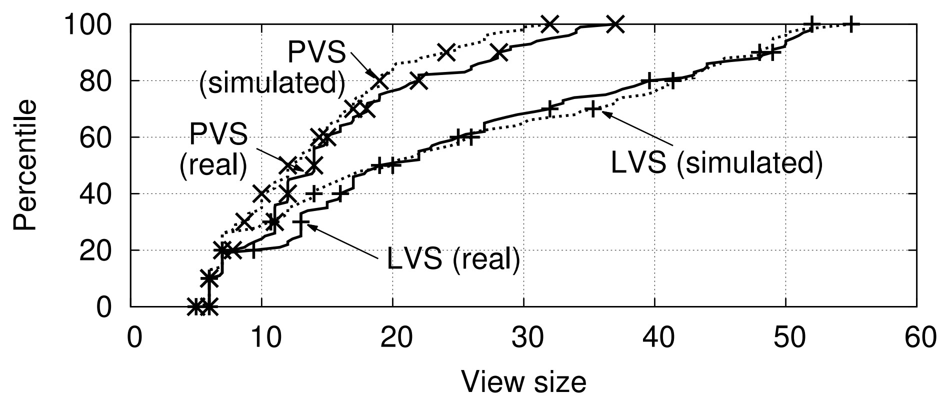
3.3.4 Performance

To assess StaN's performance in promoting link sharing we defined two measurements: *logical view size* (LVS) and *physical view size* (PVS). LVS measures the number of logical links established by a process across all topics, which is the sum of the view sizes across all process's overlays. PVS captures the number of physical links that each process needs to establish. It is obtained by extracting the unique process identifiers from the logical views. Since StaN preserves the number of logical links, we expect LVS to remain constant and PVS to decrease as the algorithm converges.

Figure 3.7(a) presents the evolution of LVS and PVS for a synthetic universe with 100 processes and 16 topics, for both the real and simulated environments. Values are averaged over 5 distinct runs. LVS remains constant across the whole experiment because StaN preserves the size of the views of individual overlays. PVS drops from 25 to around 15, which shows that StaN is able to share logical links after just a few rounds. The better performance of the simulator is explained by its discrete nature as the algorithm runs in lock step mode, and processes optimize their views before proceeding to the next round. Other reasons are the non-negligible message loss and connectivity issues experienced in PlanetLab due to faults and churn. Despite this slight deviation, one can expect

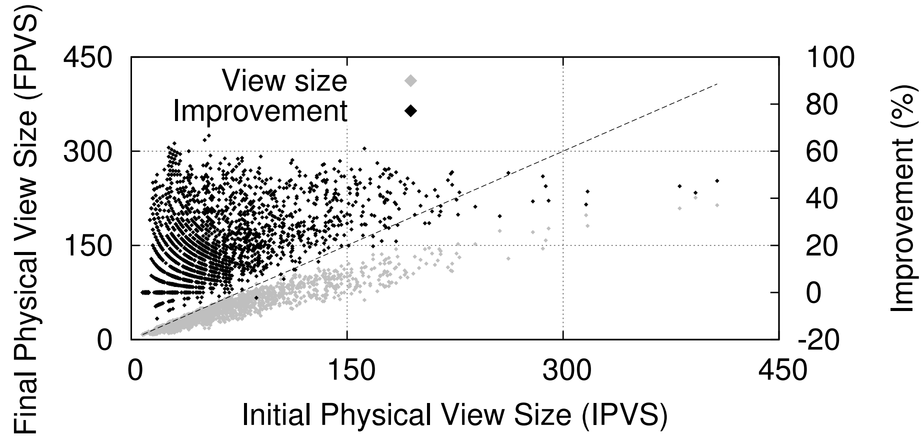


(a) View evolution.

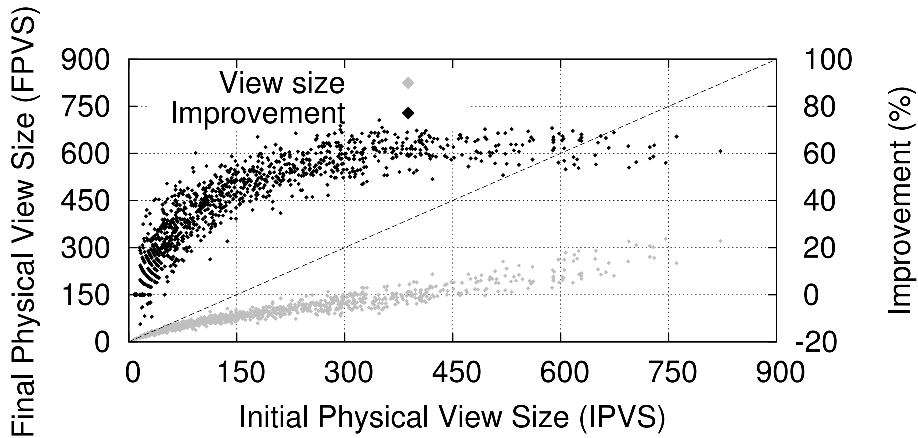


(b) View distribution.

Figure 3.7: Evolution and distribution of the LVS and PVS for a synthetic universe (100 processes and 16 topics). The legend is shared by the two graphs.



(a) $\mathcal{L}_8$ LiveJournal universe.



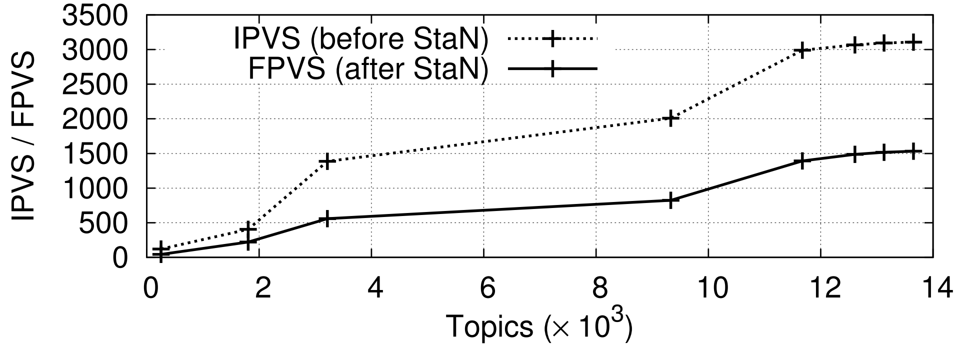
(b) $\mathcal{W}_8$ Wikipedia universe.

Figure 3.8: IPVS/FPVS and relative improvement for the $\mathcal{L}_8$ and $\mathcal{W}_8$ universes.

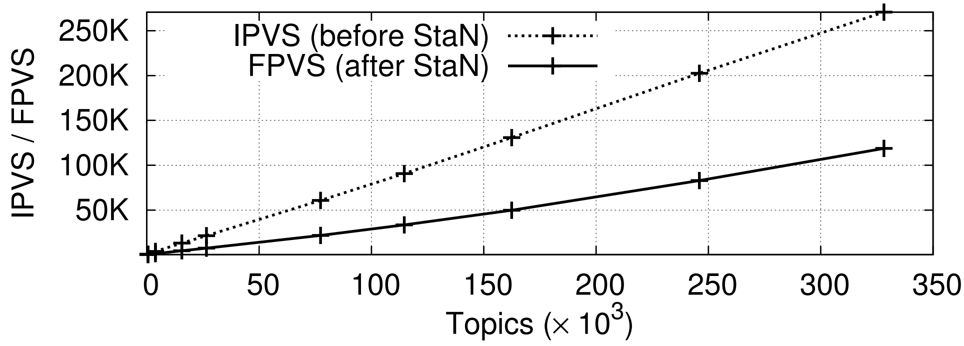
that simulation results with larger universes will be representative of real-world deployments that exceed the scale of our experimental testbed. Figure 3.7(b) presents the cumulative distribution function of the view sizes of all processes. Again the results obtained by simulation mimic those of the real environment.

From this point on, we ran StaN in the simulated environment with the real universes to study its behavior in larger scales. All results below are the average of 10 independent runs. We consider both the *initial* PVS (IPVS) at the beginning of the experiment and the *final* PVS (FPVS) after running StaN.

The next experiment aims at observing the effectiveness of StaN at reducing PVS as a function of IPVS. This allows us to detect where the improvement happens, namely to which extent processes with large view sizes benefit from



(a) LiveJournal universes.



(b) Wikipedia universes.

Figure 3.9: Evolution of the IPVS/FPVS with the number of topics.

StaN. Figure 3.8 presents these results as a scatter plot, in the x axis we have the IPVS and in the left y axis the FPVS. The vast majority of points for the view size lies below the diagonal meaning that StaN effectively reduced process's view sizes. This is confirmed when we analyze the improvement in percentage (right y axis). As expected, the improvement both in absolute and relative terms is greater for Wikipedia (bottom) due to its greater correlation. In fact, the majority of Wikipedia's processes have an improvement over 40%, while for LiveJournal fewer processes achieve such an improvement.

For some processes the improvement is negative. This only happens for processes with very low IPVS and is because the weight function of StaN can sometimes split a physical link that was initially shared (by chance) when optimizing the overlay. This may happen for all processes but it is only noticeable for processes with very low IPVS. Indeed, the number of links that become physically shared on processes with larger view sizes easily outweighs any alignment that may have occurred by chance at creation time. Results for the other universes

follow the same trend (not shown).

Finally, Figure 3.9 presents a condensed view of the previous plots for LiveJournal universes $\mathcal{L}_1 - \mathcal{L}_8$ and Wikipedia universes $\mathcal{W}_1 - \mathcal{W}_8$. The results are obtained by extracting the IPVS and FPVS for each configuration, i.e. before and after running StaN. One can observe that the number of physical links grows much slower, by a factor of two, with the number of topics when using StaN. This demonstrates that our approach is effective at scaling with the number of topics as it limits the number of physical links established by processes with many subscriptions.

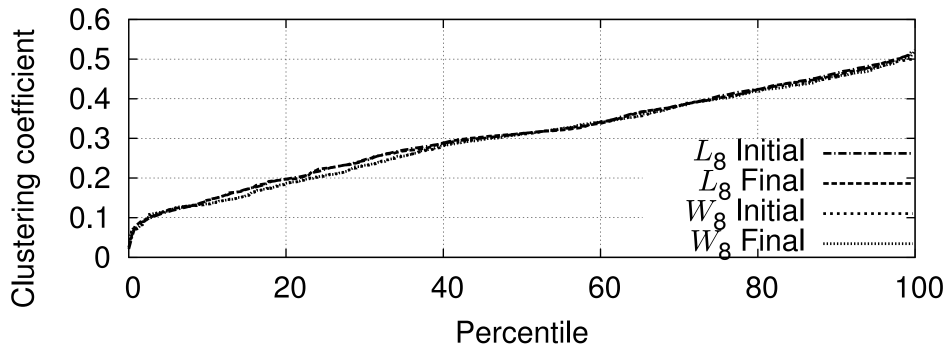
3.3.5 Fitness

We now study StaN's fitness by focusing on the structural properties of the overlays, namely *connectivity*, *clustering coefficient* and *average path length* which affect reliability and effectiveness (Jelasity et al. 2007c). Therefore, StaN must not modify them with respect to the initial values of the PSS. The experiments below are for the $\mathcal{L}_8$ and $\mathcal{W}_8$ universes, results for the other configurations are similar (not shown). Values presented are the average of each property across all overlays.

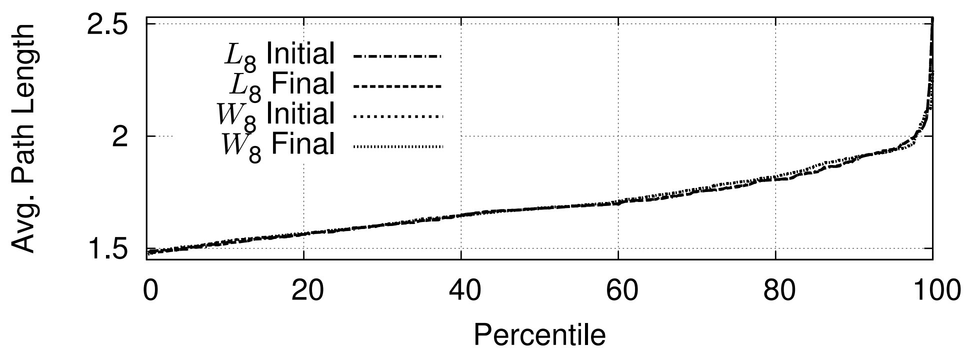
Connectivity. This property measures the number of connected components of each overlay. A single component indicates the overlay is connected. By assumption, the initial overlays are managed by a PSS that creates a single connected component. In all the experiments conducted, we did not observe a single disconnection. This is due to the uniformity of the weight function, which ensures that every process is equally likely to be selected as a best neighbor, thus compensating the loss of links when processes choose other neighbors.

Clustering coefficient. The results, presented in Figure 3.10(a), show that the initial and final values are almost indistinguishable. In fact, due to the asymmetry of the weight function, the weight any two processes assign to each other, or to a third, is completely unrelated, thus preventing processes from selecting each other as neighbors, or preferring neighbors of neighbors.

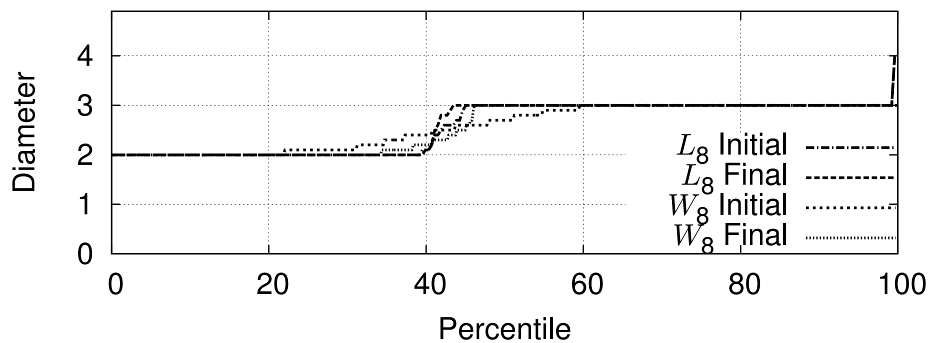
Average path length. The results, presented in Figure 3.10(b), show that, as before, both distributions are similar. The reason lies in the way the weight function is designed. As neighbors are selected uniformly at random, the probability of losing links is identical to the probability of gaining links. Moreover, the



(a) Clustering coefficient distribution.



(b) Average path lenght distribution.



(c) Diameter distribution.

Figure 3.10: Clustering coefficient, average path length and diameter distribution for the $\mathcal{L}_8$ and $\mathcal{W}_8$ universes (most lines overlap).

asymmetry prevents the overlays from converging to a grid-like structure that would otherwise increase average path length when compared to the initial random overlay. Therefore, the randomness of link establishment provided by the weight function preserves the average path length of the overlays. Figure 3.10(c) which shows the diameter - the maximum path length between any two pair of

processes - further confirms this observation.

3.3.6 Dynamics

In this section we study the behavior of StaN under conditions likely to emerge in large-scale scenarios, namely: message loss, process churn and growing scenarios where processes continue to join after the initial bootstrap. For brevity, and because the behvior under dynamics depends much more on the algorithm's design than on the particular universe chosen, we restrict the evaluation in this section to the $\mathcal{W}_8$ Wikipedia universe.

Message Loss.

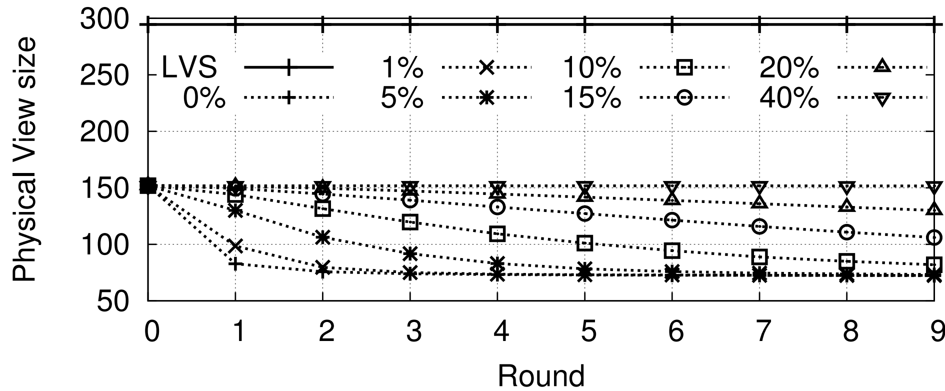


Figure 3.11: View evolution under message loss for universe $\mathcal{W}_8$ (percentages indicate message loss rates).

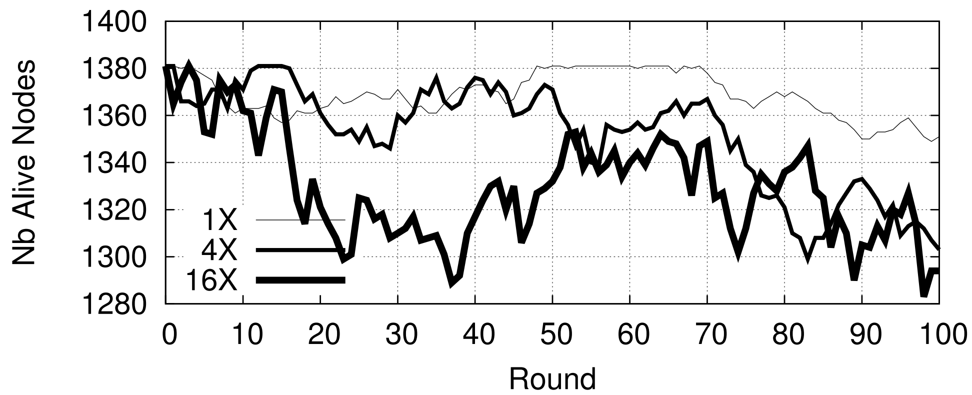
We analyze the behavior of StaN under message loss by observing its convergence speed under increasing message loss rates. This is achieved simply by dropping the given percentage of messages uniformly at random. Results can be observed in Figure 3.11. As expected, convergence speed is slowed down by message loss but StaN is still able to converge under moderate message loss rates. For instance, for a loss rate of 10%, the convergence at round 9 is almost indistinguishable from a loss-free environment. As a matter of fact, only with loss rates greater than 15%, do we observe that the convergence speed is too slow to be useful.

Process churn.

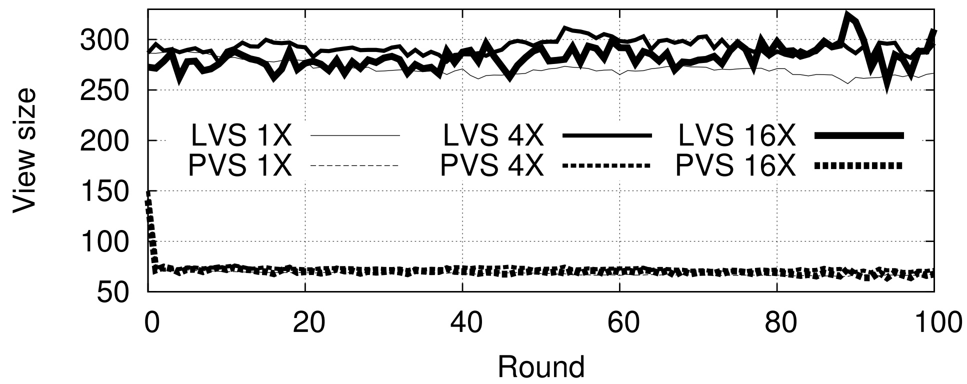
We now study STAN's behavior under process dynamics by reproducing a churn trace gathered from the Overnet network (Bhagwan et al. 2003a). For each run, we generate a trace with 1381 processes ($\mathcal{W}_8$'s universe size) and map 60 seconds of the trace time to a cycle, adding and removing processes as appropriate. We experiment with higher churn rates by *speeding up* the trace by a given factor, i.e., mapping a longer trace time to each cycle. For instance, mapping 120 seconds to a cycle yields a factor of $2X$. Figure 5.11 presents the evolution of the universe size (top) and PVS and LVS (bottom) for factors 1, 4 and 16. As expected, increasing the churn rate increases the magnitude and amplitude of the variations in the process population (Figure 3.12(a)). The same behavior is observed for the LVS and PVS which grow and shrink as the process population variates. For higher churn rates, we observe that a few processes (less than 5 in all the experiments) got isolated from the overlay. The reason is that the view size evolves only due to churn, decreasing when neighbors fail and increasing as new processes join. In some high churn cases, failures in the vicinity of one process are enough to depopulate the view without being compensated by joins, disconnecting the process from the overlay. We address this issue by triggering a random walk (Algorithm 2, COLLECTWALK()) to add new links, when the view size is smaller than a given threshold (5 in our experiments). This simple modification avoids disconnections under higher churn rates. We note that the decision to modify the view size and add links when necessary is typically the responsibility of the PSS and thus out of the scope of STAN. In this experiment this is done so that we can focus exclusively on STAN's behavior without having to be concerned how a specific PSS manages the view size.

Growing universe.

Finally, we study the behavior of STAN under a considerable universe growth. We start by randomly selecting 50% of the $\mathcal{W}_8$ processes and running STAN on that sub-universe. Then, every 10 rounds we add 10% of the remaining processes until all processes are in the system. Results are presented in Figure 3.13. When adding processes, the PVS grows quickly to accommodate the new processes which is then reduced by STAN in a few rounds. Most interestingly, some rounds after the universe is fully grown (round 60), both LVS and PVS are almost



(a) Universe size evolution under churn for universe $\mathcal{W}_8$.



(b) LVS and PVS evolution under churn for universe $\mathcal{W}_8$.

Figure 3.12: Universe and view evolution under churn for universe $\mathcal{W}_8$. (Numbers represent the churn speedup factor.)

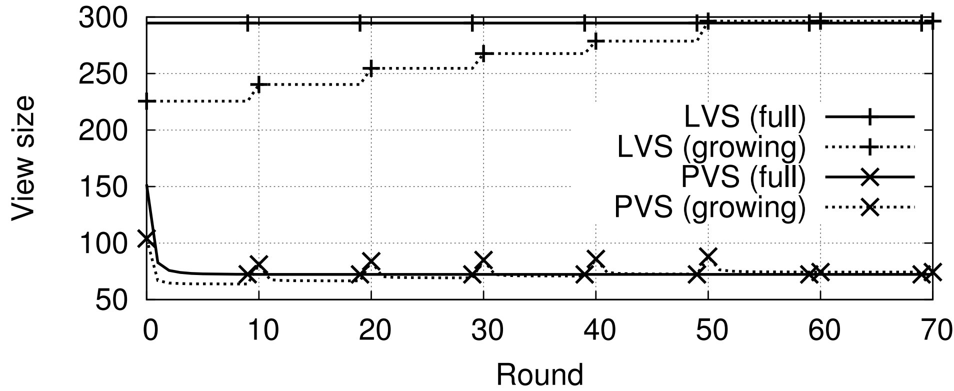


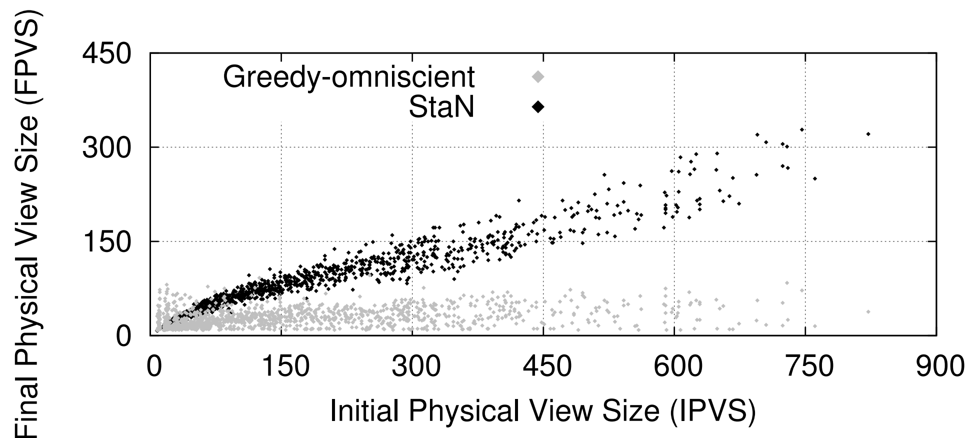
Figure 3.13: View evolution for growing $\mathcal{W}_8$ universe.

indistinguishable from a universe fully bootstrapped from scratch.

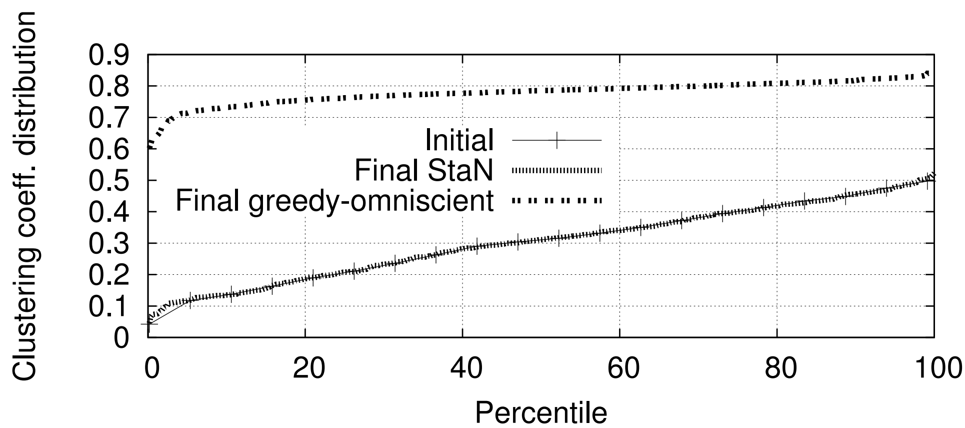
3.3.7 Greedy-omniscient comparison

StaN promotes link sharing by relying only on local knowledge and without explicitly considering process interests. We now compare StaN against a greedy-omniscient implementation with global knowledge that explicitly takes into account processes' interests to try to maximize link sharing. Our goal is to compare StaN's performance with an approach based on global knowledge and at the same time assess the impact on fitness of optimizing to an inherently clustered metric - the process's interests. The greedy-omniscient implementation works as follows: each process sorts all other processes according to the most topics in common by computing the cardinality of the intersection of its subscriptions with the other process's subscriptions. Then it picks the $viewSize$ best ones, where $viewSize$ is computed as in Section 3.3.3. Because processes have global knowledge, the process's local choices are the best possible, but as in typical greedy approaches there is no guarantee that the global solution is optimal. We note that this strategy is similar to SpiderCast (Chockler et al. 2007b) with global knowledge and with the random selection disabled ($K_r = 0$).

Figure 3.14(a) depicts the IPVS vs FPVS for both StaN and the greedy-omniscient implementation. Each point in the scatter plot represents the IPVS and FPVS for each process. As expected, the greedy-omniscient implementation outperforms StaN in terms of PVS reduction. This is because the greedy-



(a) View improvement.



(b) Clustering coefficient distribution.

Figure 3.14: Comparison of view improvement and clustering coefficient distribution for StaN and a greedy-omniscient approach for $\mathcal{W}_8$. Lines "Initial" and "Final StaN" overlap in Figure 3.14(b).

omniscient optimization criteria is precisely the reduction of the view size while in StaN the optimization criteria is a weight metric unrelated to the view size. Nonetheless, the absolute reduction in PVS obtained by StaN is still considerable. For instance, for an IPVS of 600 StaN achieves a reduction of around 400. The trade-off is increased clustering because by optimizing to the view size, the overlay tends to approximate the inherent subscriptions clustering, as observed in Figure 3.14(b). Note that lines *Initial* and *Final* StaN overlap indicating that StaN's impact on clustering is negligible.

3.3.8 Dissemination

As StaN maintains the desirable properties for epidemic dissemination (Section 3.3.5), we now study the behavior of the CrosspostFlood dissemination algorithm. This is done by analyzing bandwidth usage, in terms of number of messages exchanged, and latency, in terms of hops.

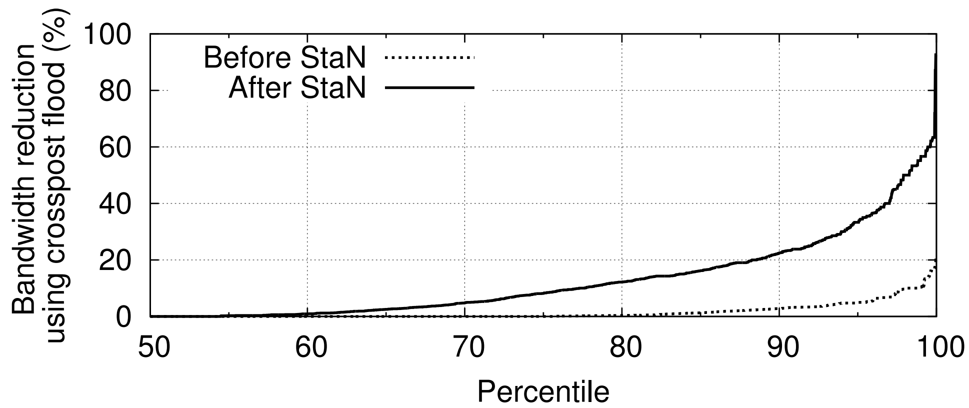
Based on real observations where, on average, each crossposted message targets 3 topics (Whittaker et al. 1998; McGlohon 2010), we devised a simple workload to compare the effectiveness of CrosspostFlood with a baseline infect and die flooding algorithm, which we call SimpleFlood.

The dissemination is done on all overlays as follows: for each topic T, we select T' and T'' as the most correlated topics with T. Next, we randomly pick 10 processes subscribed to T, T' and T'' and have each of them inject a new message on the system tagged with the triplet (T, T', T''). This is done for each of the 10 independent runs of StaN analyzed before. Results presented are thus the average of 100 independent runs (10 disseminations for each of the 10 runs) for both CrosspostFlood and SimpleFlood in the $\mathcal{L}_8$ universe.

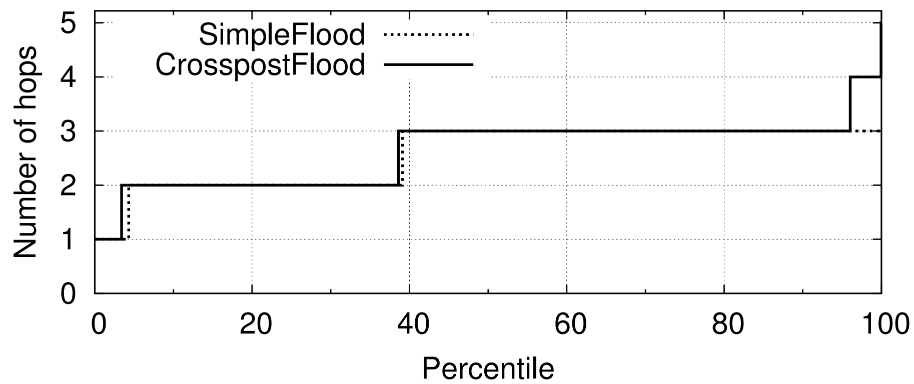
Figure 3.15(a) presents the bandwidth reduction when using CrosspostFlood and SimpleFlood before and after the optimizations performed by StaN. Results are obtained by calculating the ratio between the number of messages sent by each process using CrosspostFlood against SimpleFlood. Thus, a value of X% means that CrosspostFlood sent less X% messages overall than SimpleFlood.

It is important to note that on the worst case (no crossposting or no link sharing) CrosspostFlood degenerates to SimpleFlood. As expected, CrosspostFlood reduces the number of messages sent (the ratio is positive), thus saving bandwidth. This is more evident when disseminating after the optimizations made by StaN as there are more logical links mapped to the same physical link, thus enabling further reductions. For instance, without StaN's optimizations the amount of processes able to save (reduce) more than 10% when using CrosspostFlood is negligible. On the other hand, when using optimized overlays, more than 20% of the processes are able to achieve reductions greater than 10% when using CrosspostFlood.

At a local level, these savings are interesting as the cost is negligible: processes only need to check if several logical links map to the same physical link. To assess the cost at a global level, we need to measure the latency, in terms of number



(a) Bandwidth reduction distribution.



(b) First delivery hops distribution.

Figure 3.15: SimpleFlood vs CrosspostFlood on universe $\mathcal{L}_8$: a) bandwidth reduction before and after optimizing the overlays with StaN b) hops necessary for first delivery.

of hops needed to infect all processes. This is because reductions in bandwidth typically tend to negatively affect latency.

Figure 3.15(b) shows the hop count distribution for the reception of new messages on StaN optimized overlays. As observed, hop-counts are almost unaffected and even reduced in some situations. This is because when a crossposted message is received on a given topic, it is immediately delivered and relayed to all the relevant topics which acts as a shortcut to the normal per-topic relaying process.

3.4 Related Work

In this section we discuss some related work pertaining StaN. Approaches to topic-based publish/subscribe can be generally divided in two categories: one that maintains multiple separate overlays per topic, and other that maintains a single general overlay.

A first class of decentralized topic-based publish/subscribe rely on the structural properties of Distributed Hash Tables (DHT) to build a topic-based dissemination system. In Scribe (Castro et al. 2002), each topic is managed by a single process in the DHT, the *rendezvous* point that handles subscription and unsubscription requests. Subscribers are organized in a multicast tree rooted at the *rendezvous* process that serves as the entry point to all events. When a publisher wants to inject an event in the system, it uses the underlying DHT to discover the *rendezvous* process responsible for the target topic. The *rendezvous* process subsequently propagates the event through the multicast tree associated with the topic: all intermediate processes from the subscriber to the *rendezvous* have to relay the event, even if they are not interested in the topic. CAN-multicast (Ratnasamy et al. 2001) also associates *rendezvous* process to topics. However, each topic is managed independently with a new protocol instance. With both Scribe (Castro et al. 2002) and CAN-based multicast (Ratnasamy et al. 2001), processes that are not interested in some topic may still act as forwarders, be they internal processes of the dissemination tree built on top of the DHT or the *rendezvous* process that implements the group membership management. The existence of the *rendezvous* processes thus poses scalability and fault-tolerant concerns as the algorithms rely heavily on them. Magnet (Girdzi-

jauskas et al. 2010) relies on the same principle but uses as substrate a DHT that clusters processes according to interests. This design greatly reduces the load on forwarders, thus improving *accuracy*. StaN does not require the maintenance of a structured overlay network for its operation nor forwarder processes as it targets instead unstructured topic-based publish/subscribe overlays. These approaches require that forwarder processes not subscribed to a topic T participate in the dissemination of events pertaining to T, which is not necessary in StaN. Instead, the link sharing allows to send the same event to multiple topics by using less actual messages, while posting to several topics in Magnet imposes a linear increase of the load on forwarder-only processes. Magnet, similarly to Scribe or CAN-multicast, is more adapted to situations with a moderate number of well subscribed topics, while StaN and epidemic approaches are more adapted to a large number of topics whose popularities follow a power law distribution.

The daMulticast (Baehni et al. 2004) departs from the structured approach to embrace a pure epidemic strategy. Overlays are organized in a hierarchy, with an independent overlay per level, thus enabling completeness and accuracy. Probabilistic links are maintained from overlays at lower levels to their parents, thus reducing maintenance overhead and message complexity by exploiting the hierarchy of topics. This approach does, however, rely on these hierarchical relationships between topics. It would otherwise degrade to a traditional approach with a single overlay per topic. In contrast, StaN makes no assumptions on topic hierarchies but can nonetheless take advantage of them. As StaN works by exploiting individual processes subscriptions instead of topic relationships, it is more flexible. TERA (Baldoni et al. 2007a) relies on epidemic protocols to maintain a general overlay, used for routing, and a separate overlay per topic. One of the main goals of TERA is to provide an efficient mechanism for outer-topic routing, i.e. the ability for a process to deliver a subscription request or event to an arbitrary topic. In this sense, TERA is complementary to StaN but still suffers from scalability problems with high numbers of subscriptions as topics are maintained by fully separate overlays. Assessing whether it is possible to combine TERA's goals with those of StaN opens interesting perspectives.

To avoid the scalability problems of one overlay per topic, SpiderCast (Chockler et al. 2007b) uses a single overlay. Links are established according to two strategies: similarity among subscriptions or at random. In order to probabilis-

tically ensure topic connectivity, the protocol attempts to guarantee that each process becomes k-covered for every topic it is interested in. A process is k-covered for topic T if it has k neighbors also interested in T. The approach builds upon the theory of k-regular random graphs (Wormald 1999). Once a process becomes k-covered, SpiderCast does no longer search for processes with closer interests. With full membership knowledge, this approach works well since chosen processes are, by design, the most similar. When that is not the case, the performance degrades because the set of candidate processes may not include the most similar ones. Still, SpiderCast is able to construct connected overlays with low degree knowing only 5% of the processes. Naturally, the performance improves with the initial fraction of known processes. This is not the case, however, with a partial membership view where each process knows only a very small fraction of the system. Moreover, the decision of link addition and removal must be made by two adjacent processes due to the use of an undirected graph. The other main concern is service differentiation as it is not possible to offer different service levels based on topic requirements.

The *Min-TCO* problem (Chockler et al. 2007a) is defined as the construction of a graph with a minimum number of edges that ensures completeness and accuracy. Its decision version is shown to be NP-complete, and thus captures some inherent limitations of an approach based on a single overlay. While this results in overlays with low average view size (degree in that paper), the maximum view size can grow quite large. This is addressed by the Low-TCO (Onus and Richa 2010), which achieves both low average and maximum view sizes. However, those approaches require global knowledge, are computationally expensive and do not support subscription dynamism. Recently, these issues have been tackled by divide-and-conquer strategies that enable parallelization dynamism (Chen et al. 2010, 2011). Still, they are centralized and designing a distributed equivalent is, to the best of our knowledge, an open issue.

A survey of proposals based on subscription correlation can be found in (Querzoni 2008).

3.5 Discussion

So far, designers of topic-based publish-subscribe systems faced a dilemma: one can either manage each topic independently and pay the inherent overhead or manage all available topics in an integrated manner at the expense of fitness and accuracy. The reason for this stems from the fact that approaches trying to overcome the scalability issues of independently managing multiple topics explicitly take into account a social aspect - the interests of topic subscribers are correlated. As a result, such approaches inevitably construct overlays inherently clustered, which are undesirable in epidemic algorithms (Jelasity et al. 2007a). StaN's novelty is the departure from this explicit acknowledgement of interest overlap to a tacit exploitation of the phenomena. Similar to previous approaches, StaN computes the *best* neighbors of each process in a deterministic fashion - a condition necessary for convergence - but unlike them, the *best* neighbors are not given by their interests but by a pseudo-random weight function. This simple modification breaks the symmetry of choices, and thus effectively avoids clustering. Still, because of the implicit correlation of interests, processes end up selecting the same neighbors in several topics, and thus reduce the cost of participating in many topics by allowing links to be shared. In terms of absolute improvement on link sharing, approaches that explicitly take into account process's interests clearly outperform StaN (Figure 3.14(a)). However, the resulting clustering becomes very high and therefore the overlays are not only brittle under faults and churn, but also inefficient for epidemic dissemination. On the other hand, StaN does not affect the clustering and still provides a reduction of up to 60% in the number of physical links established. Performance is ultimately limited by the initial view size of the processes and topic correlation. As expected, the benefits of StaN are more evident in processes with large views, precisely those with scalability problems as it effectively reduces the number of physical links maintained. This is, in our opinion, an interesting trade-off in the design space - one can significantly reduce the cost of subscribing to multiple topics while maintaining the robustness and fitness for epidemic dissemination. Interestingly, this is obtained with low overhead in terms of message complexity. As a matter of fact, StaN's overhead is due to the periodic random walks. This impact is low because the TTL is small and messages only carry a small sample of process identifiers. The TTL only needs to be on the order of the overlay diameter to provide the

chance of discovering all processes. Shorter TTLs would preclude processes from opposite fringes of the overlay to know each other. Processes can control the random walk period based on the expected improvements: new processes would use small periods to quickly converge and then progressively reduce the frequency as improvements become marginal. Although not considered, processes can also leverage other process's random walks (upon a CollectWalk()) as a source of new neighbors, further reducing the impact on the network and improving convergence speed. This optimization is also useful to counter the effects of message loss and churn as processes are able to gather more information for each message delivered.

The crosspost-aware nature of CrosspostFlood when combined with the physical link sharing obtained by StaN enables improved resource usage in terms of bandwidth at virtually no local or global cost. We note that these savings are only possible due to link sharing, otherwise it degenerates to a simple flooding dissemination protocol. Moreover, CrosspostFlood is not specific to StaN as it may be combined with other protocols that promote link sharing, such as SpiderCast (Chockler et al. 2007b). The results are interesting because the improvements are obtained at virtually no cost. Nonetheless, a deeper analysis of this protocol and its combination with link sharing protocols like StaN is needed, namely by considering more complex workloads and other phenomena, such as message re-crossposting, i.e. when a process receives a message on a topic and locally decides to repost it on another topic.

The fact that StaN does not require disclosure of each process's interests - in fact it is oblivious to them - opens interesting perspectives for a privacy preserving topic-based publish-subscribe. The non-disclosure of interests adds some naive privacy preservation to StaN as honest-but-curious processes are not provided with a list of subscriptions of other processes as part of links creation and maintenance. However, honest-but-curious processes can still know many of the subscribers of the topics they belong to simply by inspecting the CollectWalk() they forward. Studying the use of cryptographic techniques in StaN to preserve the privacy of processes, strengthening the algorithm against malicious processes and researching the issue of privacy in topic-based publish-subscribe systems in general opens interesting research avenues.

Chapter 4

Brisa: efficient reliable data dissemination

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 (Gantz 2007, 2008), putting pressure on dissemination systems able to efficiently deliver the data to its intended consumers. Examples include the distribution of digital media (e.g., music, news feeds) on the Internet (Frey et al. 2009) or software updates in a datacenter infrastructure (Twitter Engineering September, 2012).

On account of its importance, significant research has been dedicated to conceiving efficient and robust data dissemination systems (Birman et al. 1999; Castro et al. 2002, 2003a; Eugster et al. 2003b; Liang et al. 2005). Unfortunately, both design vectors, efficiency and robustness, are often addressed disjointly: either by a highly efficient structure based on trees like in (Chu et al. 2002) or by a highly robust unstructured epidemic approach such as (Birman et al. 1999).

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 processes that lie in the subtree rooted at a failed process. These reconstruction delays accumulate along the path to leaves, when multiple faults occur during a dissemination further degrading dissemination latency. Furthermore, only interior processes contribute

to the dissemination effort while resources of leaf processes remain unused which leads to poor load balancing.

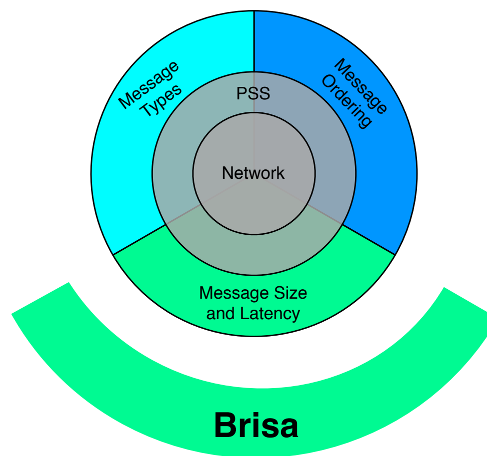


Figure 4.1: Brisa placement in the problem space.

On the other hand, epidemic dissemination systems rely on redundancy instead of structure to offer guarantees on the delivery of data to all participants (Birman et al. 1999; Eugster et al. 2003b). Epidemic dissemination was initially proposed in the context of database replica synchronization in the ClearingHouse project (Demers et al. 1987). The transmission of several copies of the same message to random processes enables epidemic systems to be oblivious to faults and churn, as the same message will be received through different paths. Epidemic principles have also been used elsewhere to build robust and scalable distributed systems components such as membership (Ganesh et al. 2001; Jelasity et al. 2007b; Leitão et al. 2007b), failure detection services (Renesse et al. 2007) or indexing mechanisms (Montresor et al. 2005; DeCandia et al. 2007b). As long as the overlay constructed by the PSS is connected, complete dissemination can be trivially achieved by flooding. The cost is increased bandwidth and processor

usage due to the transmission and processing of duplicates. As bandwidth is a limited resource, the considerable amount of duplicates received poses a problem of data scalability, i.e. these approaches do not scale with respect to message size. Even if the typical message size is moderate in relation to bandwidth, the number of duplicates sent and received can easily clog the available bandwidth and thus impair the reliability of the system. Due to this limitation, current epidemic dissemination systems are mostly used for dissemination of application control data (van Renesse et al. 2003; Renesse et al. 2007). There are however scenarios where epidemic properties are desirable, like its scalability in the number of processes, resilience to faults and churn, and load balancing among all the participants in the dissemination process, that still require the dissemination of medium to large data/message sizes in order to operate properly. Examples of this include system patches in a data center, or updates to blobs in a tuple store, with multiple messages from different sources being disseminated concurrently in the system.

Several proposals try to overcome the weakness of each approach by combining them. For instance, SplitStream (Castro et al. 2003a) builds several trees and strips 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 MON (Liang et al. 2005) and TAG (Liu and Zhou 2006) 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.

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 processes, 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. We evaluated

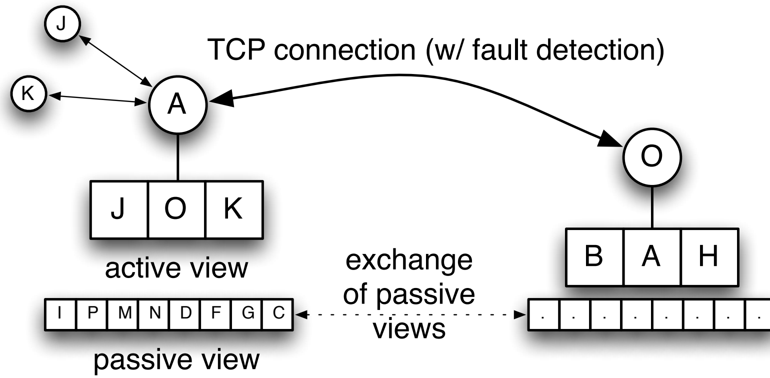


Figure 4.2: HyParView (Leitão et al. 2007b): views maintenance.

Brisa on PlanetLab (PlanetLab 2013) and on a local cluster comparing it with several data dissemination systems from the literature.

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 Algorithm description

In this section, we describe the design of Brisa. Brisa relies on an underlying peer sampling service (PSS), and thus we first discuss its requirements and the guarantees it provides. Then, we introduce 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.

4.2.1 Peer Sampling Service Layer

We assume the existence of a PSS (Jelasity et al. 2007b) with the properties discussed in Section 2.2.2 and more specifically HyParView (Leitão et al. 2007b). 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 process:

a larger *passive* view and a smaller *active* view (see Figure 4.2). Only the active view containing the process'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 process. An opened TCP connection is maintained with each of the processes 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 process is selected from the passive view and moved to the active view. When the active view is full and a new process attempts to join, a random process is removed from the active view to accommodate the joiner. In order to avoid chain reactions due to the massive number of joins when bootstrapping the system (process A's view size is full so it removes process B, B also removes A from its view and promotes a process 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*. Processes 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 4.8).

An important aspect of HyParView is that links with neighbors are *bidirectional*. If process A has process 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 processes in the system without requiring pull mechanisms - also known as anti-entropy (Demers et al. 1987) - where processes periodically poll other processes for the content they might have missed. A process receiving a message *for the first time* from a neighbor simply propagates it to all its other neighbors.

Flooding is ensured to reach all processes as long as no process in the system has an active view with only failed processes. The larger the active views the

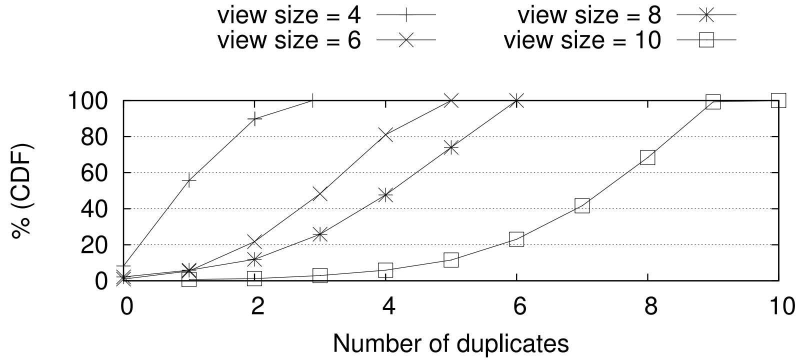


Figure 4.3: Distribution of duplicates per message for each process for 500 messages in a 512 processes HyParView network for various active view sizes.

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 4.3 presents the cumulative distribution function (CDF) of the number of duplicates during the dissemination of 500 messages over a 512 processes HyParView network for different view sizes. We observe that as the size of the view grows, processes quickly receive large amounts of duplicate messages. For instance, half of the processes 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% process failures (Leitão et al. 2007b) 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 processes (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 process 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 approaches robust, not necessarily the actual data transmission. Therefore, our goal is 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 derives 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 process to a single process (parent), or directed acyclic graphs (DAG) by allowing multiple parents for each process. The creation of a structure is performed by local and unilateral decisions made by the processes 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 processes 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 process. 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. Processes 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 process sends the first message of the stream to all its neighbors. Processes receiving the message for the first time simply forward it to all the processes in their view because all links are active, effectively flooding the network.

This flooding operation reaches all processes, given the connected and bidirectional nature of the overlay provided by HyParView. During the initial flood, processes receive the message from a number of different neighbors. Out of these sources, each process 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 process. 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 process 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 processes for reparations upon failures, or as a fallback for dissemination when reparation is temporarily not possible. Figure 4.4 presents the principle of the link deactivation mechanism for constructing a tree. Initially, links from processes X, Y, and Z belonging to process A's view and are all *active*. The first reception of a message from process X results in process A considering X as its parent. A subsequent reception of a duplicate from process Y or Z triggers the link deactivation mechanism. As only one inbound link should be active, process A needs to deactivate either the link from process X or the link from process 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,

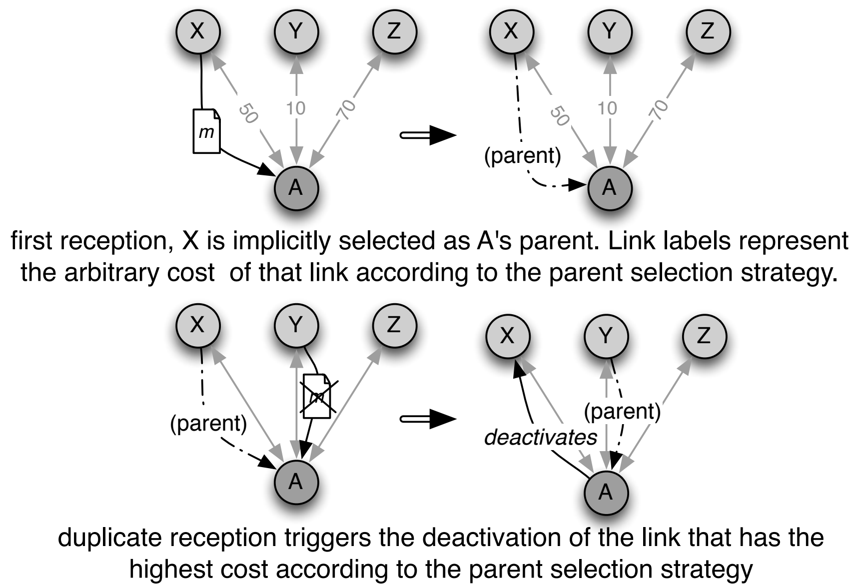


Figure 4.4: Reception of a duplicate and deactivation of one link, for a tree BRISA structure. Depending on the parent selection strategy, the deactivated link can be the previous parent or the process sending the duplicate.

the dissemination structure must not contain cycles. Second, it must seek to meet the target number of parents for each process (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 process is that it does not yield a cycle in the dissemination structure. This means that the potential parent of a process 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 process piggybacks on the application messages the process identifiers in the path from the source to itself. When selecting its parent, a process N rejects those candidates whose message path to the source includes N itself. This is illustrated in Figure 4.5, where grey processes are not eligible as parents of process N. It is important to note that the overhead of

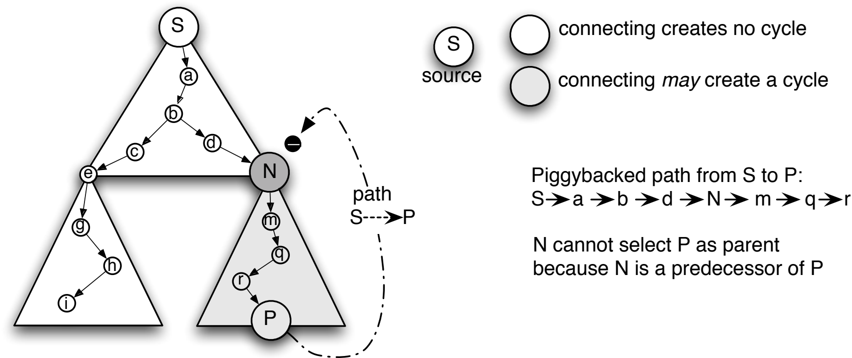


Figure 4.5: Avoiding creating a cycle for a tree, by checking that process N is not in the dissemination path to the potential parent.

path embedding is minimal and very attractive when compared to probabilistic inclusion structure such as Bloom filters (Bloom 1970). 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 N is the system size and b the active view size. For instance, in a system with 1×10^6 processes 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 significantly more bits (Bloom 1970). 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 process 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 process 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 processes. The one with the lowest delay is selected as parent. We leverage the periodic keep-alive messages that are exchanged by the processes 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 process A receives a message first from process B and then from process 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 processes in the system is handled by the underlying PSS. A new process joins by contacting a process already in the system. The new process is provided with an active view with the size of that of its contact point, and is inserted in the active views of the associated processes. Brisa automatically marks links to new processes as active. As a result, the joining process 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 process failures is performed at the level of the active view, by exchanging periodic keep-alive messages over the established TCP connections, or when a process fails to acknowledge the reception of a transmission (as detected by the TCP flow control for that link). When a process 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 process 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 process first re-activates all its incoming links and considers itself a fresh process by forgetting its position in the cycle detection mechanism. This allows the orphan process 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 process. The need to repair a portion of the tree is detected by the children of the orphan process when they receive an activation request from their (former) parent. Those processes 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 process 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 processes affected by a hard repair is independent of the position of the original orphan process in the tree: it only depends on processes in the sub-tree finding a suitable replacement parent, which is independent of the position of the original orphaned process.

Finally, processes can compensate message loss during recovery by directly asking its new 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 (Koldehofe 2003) could still be used to ensure processes 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 important to note that PSSs in general, and HyParView in particular, are very robust to disconnections and able to maintain

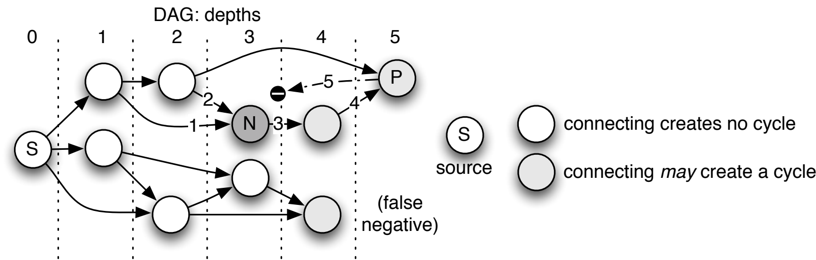


Figure 4.6: Avoiding creating a cycle for a DAG, by checking that the level of the potential parent is less than or equal to the level of the process.

connectivity even under massive failures (Leitão et al. 2007b). 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 process being served by several parents instead of only one. In this way, a process 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 process in the nth level of the tree requires a set of n process identifiers to define the path from the stream source to itself, while for a DAG with p parents per process 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 processes identifiers but just the *depth* each process is in the DAG as illustrated by Figure 4.6. The source process is at depth 0 and every message carries its sender's depth encoded by a single integer. Initially, the depth of a process N is undefined and, upon reception of its first message from a process with depth $i - 1$, N places itself at depth i. From then on, N

can select parents, and thus receive messages, from processes at any depth not greater than i. Should N receive a message from a process at depth i (its current depth) then N moves to depth $i+1$ and immediately updates its downstream children processes accordingly. Similar to the technique we used for trees, it is clear that any process M served directly or transitively by process 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 4.6. Any two paths (rooted at S) are likely to be labeled similarly with respect to depths. Since the tagging is purely quantitative, a process from one path may be dismissed as a potential parent of a process in another path despite the paths being causally unrelated. An alternative is to rely on Bloom filters to maintain the set of processes 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, processes 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 (Castro et al. 2003a) 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 processes can contribute to the dissemination effort. This is because when using a single tree, the leaf processes,

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* process that acts on behalf of all sources as in Scribe (**?**). This design suffers however from a bottleneck in the *rendezvous* process and fails to take advantage of the upload bandwidth available at leaf processes.

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 process locally maintains. Consequently, all that is required to maintain multiple trees is to locally maintain multiple such sets, one for each tree in the system. To this end, each tree is uniquely identified by a *flowId* generated by the tree source at construction time. Note that as, by assumption, each process has a unique id, it is straightforward to generate unique *flowIds*, for instance by concatenating the process id with a local sequence number. The source then tags all application messages with its *flowId*, enabling other processes to uniquely assign the messages to the appropriate tree. Upon reception of a message from an unknown *flowId*, a process 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 processes 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 process 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 process simply reuses one it already knows to disseminate its messages. To this end, the process analyzes 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 processes 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 process will simply be relayed through the existing tree with no further overhead. However, as the source process is not located at the root of the tree anymore, it is necessary to

relay messages upward in the tree, to ensure completeness. This is easily achieved by adding an *upward* flag to the message, implying that processes 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* process. We note that the latter shows less bottlenecks problems than Scribe as it considers several *rendezvous* processes, 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 (Castro et al. 2003a) or as shown in our evaluation Section 4.3. 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 processes 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 the dissemination load is unbalanced. Nonetheless, tree reusing can still be beneficial, between trees that are used for different streams. Therefore, 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 processes and (2) a slice of up to 200 processes on the global-scale PlanetLab (PlanetLab 2013) testbed. Similarly to StaN the prototype also leverages Splay (Leonini et al. 2009), an integrated system for the development, deployment and evaluation of distributed applications (see Section 3.3.3 for more details).

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 processes using the first-come first-picked strategy with an expansion factor of two, randomly choose a process to be the source across all the experiment and then have it inject 500 messages at a rate of

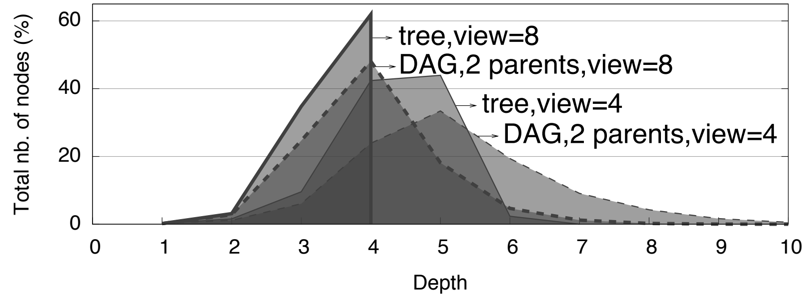


Figure 4.7: Depth distribution for 512 process (first-come first-picked strategy).

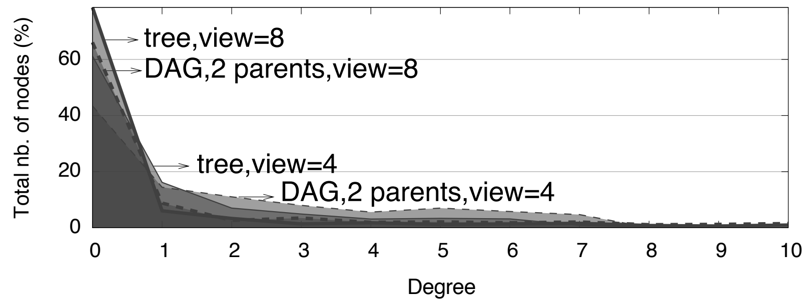


Figure 4.8: Degree distribution for 512 process (first-come first-picked strategy).

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

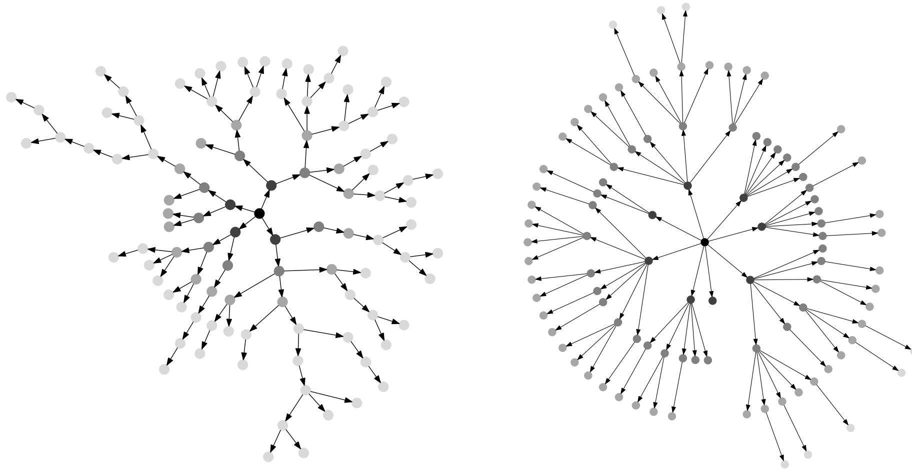


Figure 4.9: Sample tree shape for 100 processes represented in a radial layout. The HyParView active view size of 4 (left) and 8 (right). Expansion factor is 1.

stabilize completely. The reason for using the basic first-come first-served 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 processes hides significant differences on the observation of structural properties changes that are better observed at larger scales. For instance for a perfect binary tree doubling the number of processes only increases the maximum depth by one. Depth places a lower bound on the dissemination time due to the cost of traversing several intermediate processes and thus should be kept as low as possible. Figure 4.7 presents the depth density distribution in a universe with 512 processes. As expected, larger views allow processes 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 process, 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 process in Brisa is given by the number of outgoing links and thus bounds the number of messages a process needs to transmit. 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 4.8 three main observations arise. First DAGs are more effective than trees in having a greater share of the processes contribute to the dissemination effort (processes 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 4.9, which depicts sample trees obtained by Brisa. As a matter of fact, despite using a simple strategy, the resulting trees are fairly balanced which is essential for efficient dissemination. Finally, despite using an *expansion factor* of 2 the number of processes with degree higher than the configured value 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 on the PlanetLab testbed. To this end, we use the cumulative round trip times, taken at each hop, from the root to a given process. When compared against the round trip time of direct communication between the root and that process, 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 processes, we instead present the cumulative distribution of the raw results in Figure 4.10. 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, 40% of the processes reduce the routing delays to half.

Next, we focus on bandwidth usage. This measures the effort and is directly influenced by the depth and degree distribution. Figure 4.11 and Figure 4.12 depict download and upload bandwidth usage, respectively, for payloads of 1, 10,

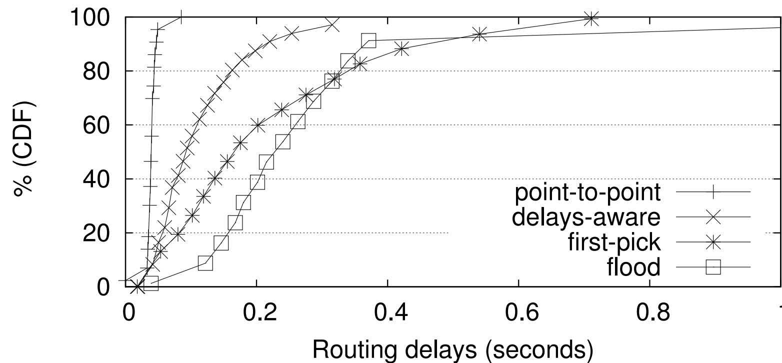


Figure 4.10: Routing delays distribution on PlanetLab for 150 processes. Structure is a tree with view size 4. Message size is 1KB×200 messages.

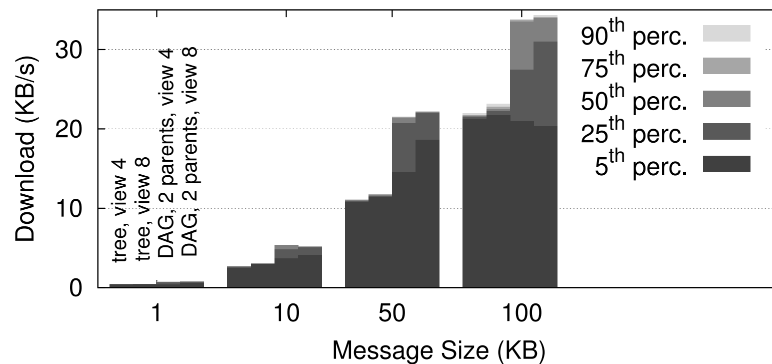


Figure 4.11: Download bandwidth usage for 512 processes.

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 processes below that value, the other half above), while the lighter shade gives the 90th percentile: 90% of the processes are associated with a lower bandwidth.

As expected, trees are more frugal with respect to download as processes receive exactly one copy of each message whereas in DAGs processes 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 processes (Figure 4.7) as processes at lower depths may not be able to find additional parents and thus receive messages only from a single parent.

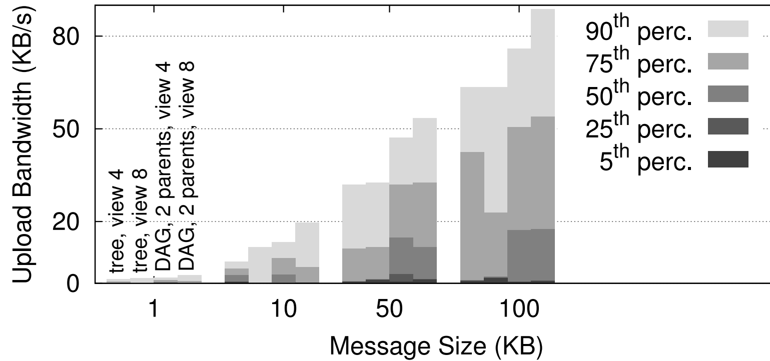


Figure 4.12: Upload bandwidth usage for 512 processes.

For upload, results are naturally similar. DAGs require more links and consequently processes 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 4.8) as processes 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 processes fail at random and X percent new processes join the system during each minute.

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

Listing 4.1: Splay's churn trace generation script.

Table 4.1 presents the results obtained for networks with 128 and 512 processes. For simplicity we ensure that the source process does not fail. However, we note that the failure of source process 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 processes lose any of their parents;
- **Orphans per minute:** rate at which processes lose all parents, i.e. become disconnected;
- **Percentage of soft repairs:** upon disconnections, how many processes successfully repair their incoming links using the *soft repair* mechanism;
- **Percentage of hard repairs:** upon disconnections, how many processes 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 processes being seldom fully disconnected. For instance, with a churn rate of 5% per minute, which implies half of the processes leaving the system within the ten minutes of the experiment, only 17 processes 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 processes are able to promptly repair connectivity with minimal effort. Finally, quick parent recovery also allows processes 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 very large scale scenarios.

4.3.4 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 in 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 processes. Figure 4.13 depicts the number of trees where

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

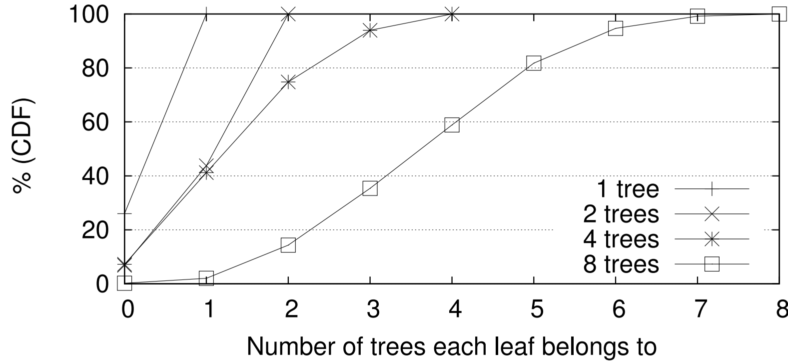


Figure 4.13: Distribution of the number of trees where processes are leaves for 512 processes and active view size of 8.

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

This is confirmed in Figure 4.14 which presents the number of children of each process across all trees. As is it possible to observe, the number of processes 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,

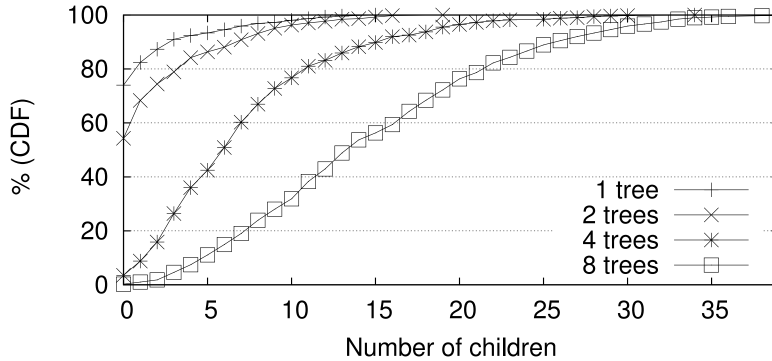


Figure 4.14: Distribution of the number of children across all trees for 512 processes and active view size of 8.

almost 80% of the processes do not upload - they have no children - 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 processes (Section 4.2.8). We note that this is achieved without explicit coordination among processes or by using more complex mechanism as in SplitStream (Castro et al. 2003a). 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 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 processes, and gives the compounded 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 processes: upon reception of a message, a process 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 process 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 processes, but also the individual performance of multiple trees is similar to that of a single tree. We account for

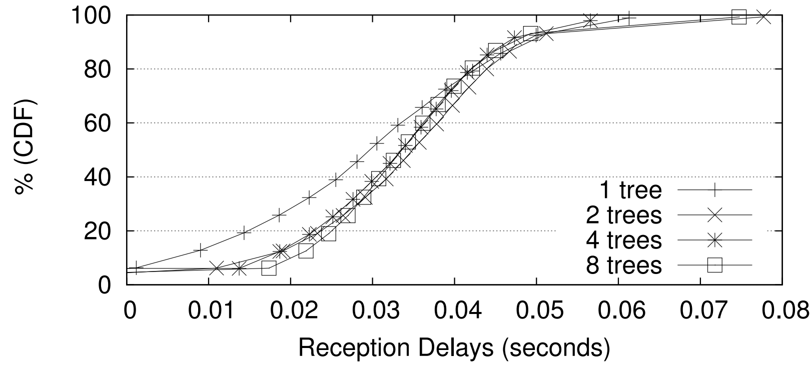


Figure 4.15: Reception delays per message when using multiple trees for 512 processes and active view size of 8. The number of messages is 500.

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 processes' 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 processes, the dissemination delay measures the time it takes for a process 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 process, which essentially gives the upload requirement, enabling an application designer to choose the right amount of trees tolerated by the underlying physical network.

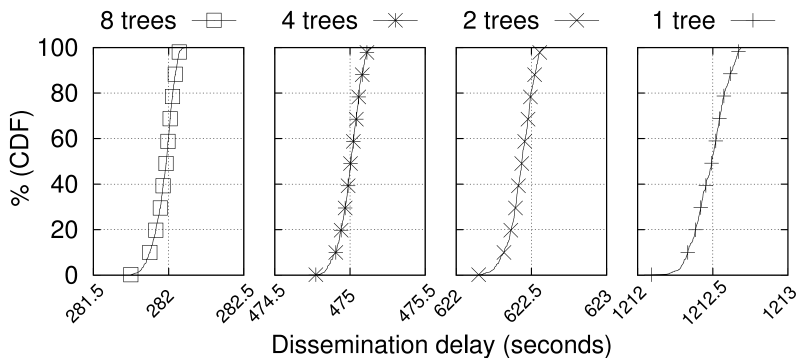


Figure 4.16: Dissemination delay when splitting the stream of messages across multiple trees for 512 processes and active view size of 8. The number of messages is 500.

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 algorithms. The algorithms 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 algorithms we considered are the following:

SimpleGossip This approach lies on the robustness end of the spectrum. We use Cyclon (Voulgaris et al. 2005b) as the PSS. Due to its proactive nature we use

a combination of rumor mongering (push) to infect most of the processes and anti-entropy (pull) to ensure completeness (Demers et al. 1987). 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 process. The only criteria for a process 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 below. This parent is provided by the centralized process that randomly picks any of the previously joined processes as a parent for a newly joined process. Dissemination is done by pushing the messages immediately through tree links thus minimizing latency.

Tag The approach use in TAG (Liu and Zhou 2006) tries to achieve both robustness and efficiency at the same time sharing thus similar goals to BRISA. As BRISA, TAG maintains a tree and an unstructured overlay to combine the efficiency of trees and robustness of epidemics. Processes are further organized in a linked list sorted by joining time, with processes maintaining information about their predecessors/successors up to two hops away. New processes traverse this list backwards until an application specific condition is met. In the traversal, processes pick k random peers to form the overlay and join the tree by choosing a suitable parent. Upon parent failures, processes update the linked list and traverse it to find a new parent and thus restore the tree. Regarding dissemination, TAG uses a pull-based approach with processes 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 epidemic robustness) and the differences in its design choices (a different 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 algorithm by considering two metrics: *stabilization bandwidth* and *dissemination bandwidth*.

Stabilization bandwidth is the bandwidth used to bootstrap the algorithm 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 processes. 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 processes. 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 process is needed while the other algorithms require inter-process 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 process 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 process joins the

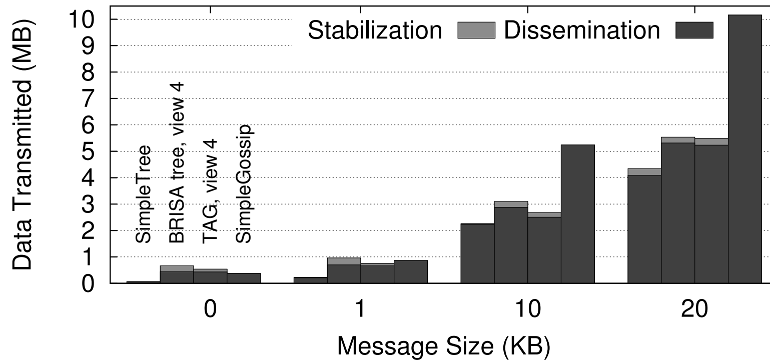


Figure 4.17: Comparison of bandwidth usage for 512 processes.

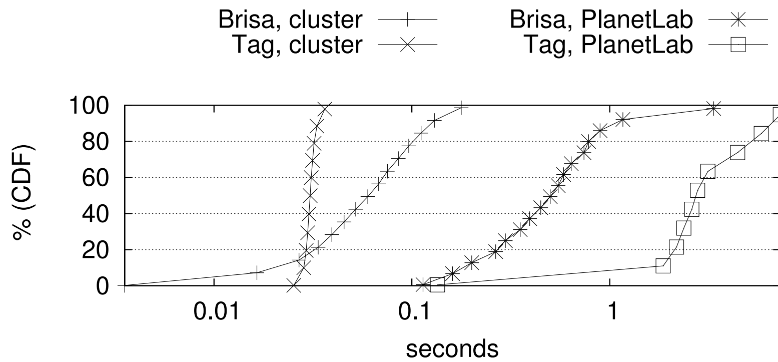


Figure 4.18: Construction time for 512 (on cluster) and 200 (PlanetLab) processes. X axis is logarithmic.

list until it settles its position on that 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 process, exchanging messages, tearing it down and proceeding to the next process) easily outweighs the time Brisa needs to wait for the reception of the messages from all its neighbors.

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 512 processes for 500 messages of 1KB.

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 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 processes. 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 a SimpleTree, and largely in favor of Brisa when using a delay-aware selection strategy as previously illustrated in Figure 4.10. 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 algorithms 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 replace-

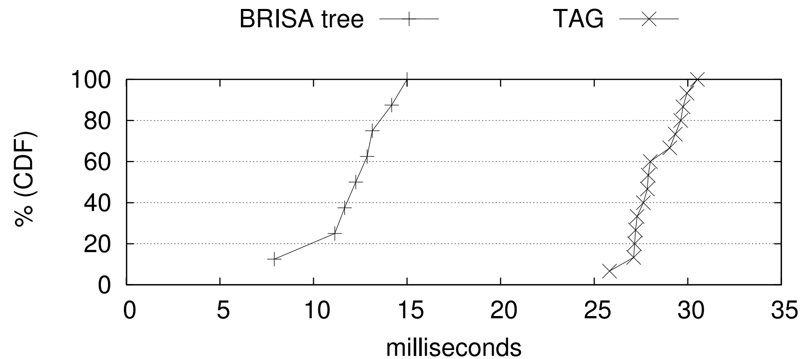


Figure 4.19: Parent recovery delays for 128 processes with active view size 4 under 3% continuous churn.

ment neighbor is available and the unstructured overlay is used instead. In TAG this corresponds to the case where the linked list is broken (i.e., two consecutive simultaneous process failures) and the process needs to be re-inserted into the structure. Figure 4.19 depicts the results for 128 processes. 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 BRISA.

Scribe (Castro et al. 2002) is an application-level multicast layer that builds dissemination trees by aggregating reverse paths to a rendezvous node in the Pastry (Rowstron and Druschel 2001) distributed hash table (DHT). Unlike BRISA, where we assume that all nodes are interested in all messages, Scribe supports group membership management by having each node subscribe 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 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 (Baldoni et al. 2007b). These group specific views can themselves be constructed by the means of an epidemic clustering protocol such as T-Man (Jelasity et al. 2009).

SplitStream (Castro et al. 2003a) is a high-bandwidth dissemination layer built on top of Scribe (**?**) and Pastry (Rowstron and Druschel 2001). In order to balance the load of dissemination, it constructs multiple Scribe trees that are used for sending alternate pieces of a stream. Nodes participating as a leaf in one tree are placed as interior nodes in other(s) trees, 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 (Venkataraman et al. 2006) 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 (Kostic et al. 2003), 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 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 dissemination mechanism to support failures. Rappel (Patel et al. 2009b) is another example

of a dissemination service that combines a tree structure for dissemination with an epidemic service for optimization purposes. In the case of Rappel, the epidemic layer is used to locate suitable peers based on interest-affinity and network distances, and not as a fallback mechanism for dissemination.

MON (Liang et al. 2005) 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 (Voulgaris and van Steen 2007) stems from an observation similar to ours that even though epidemic 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 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 (Li et al. 2008, 2011) 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 structure construction.

In (Fei and Yang 2007) 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 known 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 delay. Brisa uses a notion similar to

the alternative parents without however having the tree degenerate into a chain, neither requiring a pre-computation of the suitable alternative parents. This is because (Fei and Yang 2007) 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 (Tang et al. 2005) builds a dissemination tree embedded on an epidemic 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 (Leitão et al. 2007a) and used to recover missing messages due to tree disruptions that, contrary to Brisa, imposes 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 overlay. More details can be found in Section 4.3.5.

Similarly to Brisa, PlumTree (Leitão et al. 2007a) also relies on the detection of duplicates and subsequent deactivation of links to build an embedded spanning tree on an unstructured overlay built by HyParView. 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 missing payload the tree repair mechanism is triggered. This approach is sensitive to variations in network latency, which lead to unnecessary message recoveries as observed in (Ferreira et al. 2010). Brisa does not separate the dissemination of the metadata and the payload: the dissemination is deterministic (through the active links), and faults are detected at the underlying PSS layer. In this way Brisa removes the need for sending periodic probe messages at the level of the dissemination layer; avoids the complexity of managing timers for recovery purposes and removes the overhead of the continuous exchange of message identifiers. 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 sparingly sent to conserve bandwidth with an impact on recovery time, or advertisements are aggressively sent imposing a constant management overhead in the system. To cope with multiple senders, PlumTree uses two approaches: a single, multi-source tree, or multiple trees by sender. The first approach is similar to BRISA's tree reusing and works because the links established are bidirectional allowing a message created by a leaf to reach everyone. The downside, similarly to BRISA's tree reusing, is a considerable penalty in end-to-end latency. Alternatively, when using multiple trees, one per source, the control overhead in the form of lazy push messages increases proportionally to the number of the trees. In the optimized version of the protocol with multiple senders this control overhead is 20% higher than in the non-optimized version. In BRISA there is no network overhead for the existence of multiple trees on the system, nodes only need to maintain a local structure with which links are active on each tree.

Thicket (Ferreira et al. 2010) 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. 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 Discussion

Data dissemination is a crucial problem in distributed systems as the huge body of existing research attests. When developing a data dissemination system, designers forcibly need to consider two key aspects and the tradeoff they yield: efficiency and reliability. Strikingly, the need to excel at the two ends of the spectrum is constantly increasing. On one hand, the exponentially increase in the amount of data produced and exchanged (Gantz 2007, 2008) demands highly efficient systems, while in the other hand the very large scale of modern systems - and consequently the continuous occurrence of faults and churn (Schroeder and Gibson 2007; Schroeder et al. 2009; Verespej and Pasquale 2011) - call for a robust data dissemination system. Brisa addresses these key aspects by decoupling them in two different components. The epidemic unstructured overlay guarantees robustness and scalability even under faults and churn acting as a safety net for the dissemination. On top of that, Brisa judiciously builds a structure used for the bulk of dissemination thus removing the overhead of traditional epidemic dissemination. The combination of both components leverages the key observation that the fundamental requirement for robustness is the *possibility* of receiving duplicates, not the actual data transmission. With this in mind, the path diversity that naturally exists in unstructured overlays is kept dormant most of the times - through the inactive links - and only used when strictly necessary to build or repair the dissemination structure which is the key for efficiency. Because inactive links can be promptly reactivated when necessary, the dissemination structures can be quickly repaired (Section 4.3.3) and involving only the PSS neighbors of the process affected by the failure thus overcoming the major hurdle of traditional structured approaches.

Using dissemination structures such as trees has also the disadvantage of skewed load balancing because leaf processes do not contribute to the dissemination. This is usually overcome by using multiple dissemination structures and parallelizing the dissemination among them as done in SplitStream (Castro et al. 2003a) or Thicket (Ferreira et al. 2010). However, when disseminating through multiple trees, the effect of failures is amplified because a single failed process can compromise delivery in several trees. As such, approaches such as SplitStream and Thicket take special care to build trees where, ideally, processes are interior in just a single tree and leafs in all the others. As a complement, it is possible

to use network coding techniques in the disseminated data such that the loss of some messages - for instance due to a failure - can still be masked by the redundant information carried on other messages from different trees (Mea et al. 2007; Nguyen et al. 2010). In our opinion, the main motivation behind these techniques can be pinpointed again to the difficulty in building, maintaining and repairing the dissemination structures. Because of this, in Brisa the concern of having processes participate as interior nodes in several trees is much more relevant for load balancing purposes than for the continuity and completeness of the dissemination itself. As such, it is enough to rely on the inherent randomness of the physical network and on the PSS to support multiple trees with reasonable load balancing properties (Section 4.3.4).

Brisa's parent selection strategy allows the construction of dissemination structures with different performance criteria (Section 4.2.5). With this in mind there are several other strategies that can be considered to select the *best* parent. We highlight some possibilities: i) *gerontocratic:* which takes into account the uptime of the candidate processes and selects the one with the highest value. This is based on the observation that the higher the uptime of a node, the more likely it is to remain available (Bhagwan et al. 2003b), ii) *heterogeneity-aware:* which considers the available bandwidth at candidate processes and iii) *load-balancing:* which selects parents according to load and is, to some extent, the dual of the gerontocratic strategy by spreading the dissemination effort to newer processes. However, it is important to note that the effectiveness of these strategies is limited in practice by two factors: the bounded size of the PSS active view and its reactive nature. As a matter of fact, because all processes currently have the same active view size it is not possible to fully exploit a strategy such as *heterogeneity-aware* because a high capacity process can serve at most the same number of processes than a low capacity process. Moreover, because the active view is built randomly, and thus oblivious to any performance criteria, the effectiveness of more elaborate strategies as the ones identified above is limited. Finally, because the active view is reactive, new processes joining the system, which are potential *best* parents to some existing processes might not be taken into account. This can be addressed by building the active view of the PSS taking into account the parent selection strategy instead of at random while keeping the same properties. As an initial approach, one can leverage a topology construction protocol like T-Man (Jelasity

et al. 2009) to build and maintain the active views and thus offer better support to more adequate parent selection strategies.

The frugality and robustness of Brisa, combined with the low end-to-end delay of its dissemination structures, and the ability to quickly repair failures led us to consider using Brisa in more demanding dissemination scenarios and in particular live streaming video dissemination. This work which already takes into account the issues of more efficient parent selection strategies, is submitted at the time of this writing to a conference as *LayStream: A Layered Approach to Gossip-based Live Streaming. Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, and Etienne Rivière.*

Chapter 5

EpTO: epidemic total order dissemination

5.1 Introduction

The ordering of events is one of the most fundamental problems in distributed systems and as such a large body of research has been dedicated to the design of ordering abstractions with different guarantees and trade-offs (Lamport 1978; Défago et al. 2004). Most of those abstractions focus on providing strong deterministic guarantees that enable distributed applications to solve various problems, such as synchronization, agreement or state machine replication. Unfortunately, such strong guarantees are expensive to obtain and the algorithms that implement them are known to scale poorly, resulting in a mismatch between what is expected by distributed applications and the properties achievable in the large-scale systems encountered in the real world (Birman et al. 1999; Vogels 2009). Indeed, large-scale systems are prone to failures and partitions, and one typically has to relax some guarantees. Notably, the so-called CAP theorem (Brewer 2000; Gilbert and Lynch 2002) states that one cannot achieve at the same time consistency, availability, and partition tolerance. Therefore, many practical systems provide some degraded form of these properties, the best known being *eventual consistency* (Vogels 2009), which states that the system will reach a consistent state given a sufficiently long period of time over which no changes occur.

In this chapter, we focus on the problem of data dissemination with reliability and ordering properties. We argue that, in large-scale settings, it is impractical to

strive for strong deterministic guarantees as it would lead to prohibitively expensive or non-scalable algorithms. As a matter of fact, this is the same reason that led to the emergence of probabilistic dissemination algorithms based on epidemic principles as an alternative to deterministic dissemination algorithms (Dan et al. 1987; Hayden and Birman 1996; Birman et al. 1999; Koldehofe 2002; Eugster et al. 2003b; Carvalho et al. 2007). Such algorithms provide probabilistic guarantees, typically reaching convergence *with high probability* in a finite time (*eventually*). Additionally, they are known for their extreme scalability and resilience to churn, which are desirable properties in a real world deployment. However, most existing epidemic dissemination protocols have focused exclusively on the (probabilistic) reliability of the dissemination, overlooking stronger properties, such as ordering. Unfortunately, the absence of ordering properties, and in particular total order, limits the usage of epidemic dissemination algorithms in a wider range of scenarios.

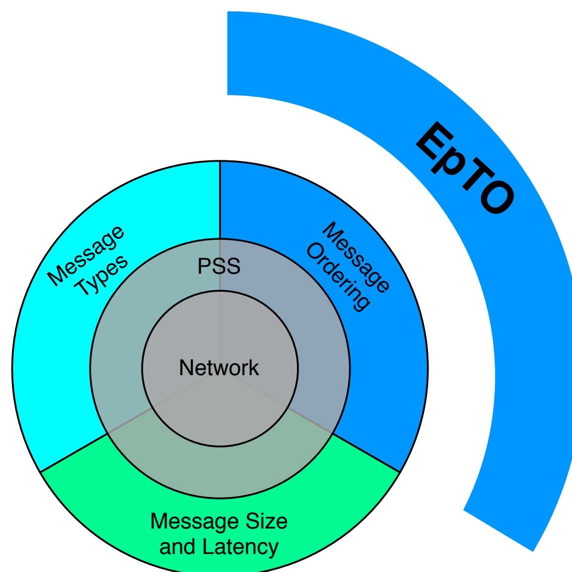


Figure 5.1: EpTO placement in the problem space.

To help overcome this problem, we propose a new epidemic dissemination algorithm, EpTO, with ordering and probabilistic reliability guarantees. It guar-

antees that processes eventually agree on the set of received events w.h.p. and deliver these events in total order to the application. The core of the algorithm is precisely on determining how long shall one wait without requiring coordination or synchrony assumptions. Note that this is substantially different from existing optimistic total order algorithms (Sousa et al. 2002; Saito and Shapiro 2005) based on spontaneous order - the physical order in which messages are received from the network - because they still require processes to agree on a definitive final order, and thus rests on deterministic reliable dissemination. Instead, EpTO relies solely on probabilistic dissemination, and thus does not suffer from the same limitations.

We start with a *balls-and-bins* approach to dissemination (Koldehofe 2002) and for simplicity assume that processes have access to *global time*, e.g. as provided by a GPS or atomic clock and used by Google's Spanner (Corbett et al. 2012). In a second step, we lift the assumption of global time and show how one can rely just on logical time. This extension does not require modifications to EpTO but only adjustments to an oracle able to inform about the deliverability of events.

We finally present extensive evaluations of the behavior of EpTO in realistic settings. Results indicate that EpTO scales well with the number of processes and events, and achieves agreement with high probability in all but the most extreme scenarios.

The rest of the paper is organized as follows: in Section 5.2 we define the system model, state the problem, present EpTO and discuss its properties. In Section 5.3 we evaluate EpTO under different realistic conditions and in Section 5.4 we discuss related work. Finally, Section 5.5 concludes the chapter and points toward future work.

5.2 Algorithm Description

In this section we start by presenting our assumptions, describing the system model, and precisely defining the problem. Then, we present the rationale behind EpTO, describe it in detail and discuss its properties.

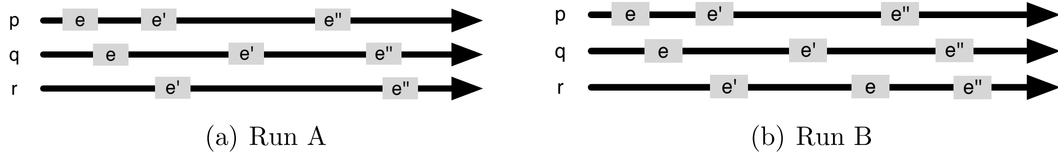


Figure 5.2: Properties satisfiability: order but no agreement (left) and agreement but no order (right).

5.2.1 System model and assumptions

As in StaN and Brisa, we assume that processes have access to a peer sampling service providing uniform random sample of other processes (Section 2.2.2). Moreover, we initially assume that processes have access to a global clock. We will then drop this assumption and, instead, only require processes to have access to a local clock. Local clocks are not necessarily synchronized but we assume that their drift is bounded.

5.2.2 Problem Statement

We are interested in data dissemination with reliability and ordering guarantees. In particular, we want to ensure that any broadcast event is delivered with high probability to all correct processes and that the very same order of events is observed at all recipients.

We consider that processes use primitives EpTO-broadcasts and EpTO-deliver to communicate and that the following properties are satisfied:

Integrity: For any event e, every process EpTO-delivers e at most once, and only if e was previously EpTO-broadcast.

Validity: If a correct process EpTO-broadcasts an event e, then it eventually EpTO-delivers e.

Total Order: If processes p and q both EpTO-deliver events e and e', then p EpTO-delivers e before e' if and only if q EpTO-delivers e before e'.

Probabilistic Agreement: If a process EpTO-delivers an event e, then with high probability all correct processes eventually EpTO-deliver e.

Besides agreement, which is probabilistic, the remaining properties closely follow those from traditional total order (or atomic) broadcast algorithms (Défago et al. 2004). The integrity property precludes spurious messages by disallowing

the delivery of duplicates and messages not previously sent. Liveness of the protocol is ensured by the validity property that requires correct processes to always deliver the messages they broadcast.

While the total order property is standard, its interplay with the probabilistic agreement guarantees of the protocol is of particular interest. Since the protocol reliability is probabilistic, holes may occur (although with low probability) in the sequence of messages delivered at each process. While these sequences may differ for any pair of processes, the order of any two delivered messages should be the same. Consider, for instance, the two runs depicted in Figure 5.2 with three processes and three events. In the run on the left (Figure 5.2(a)) the total order property is preserved but agreement is violated because process r did not receive event *e'*. Therefore, this is a valid, although unlikely, run in EpTO. On the other hand, in the run on the right (Figure 5.2(b)) agreement is preserved (all processes received all events), but total order is violated because process r delivered the events in a different order from the other two processes. Consequently, this run is not allowed in EpTO.

5.2.3 Rationale

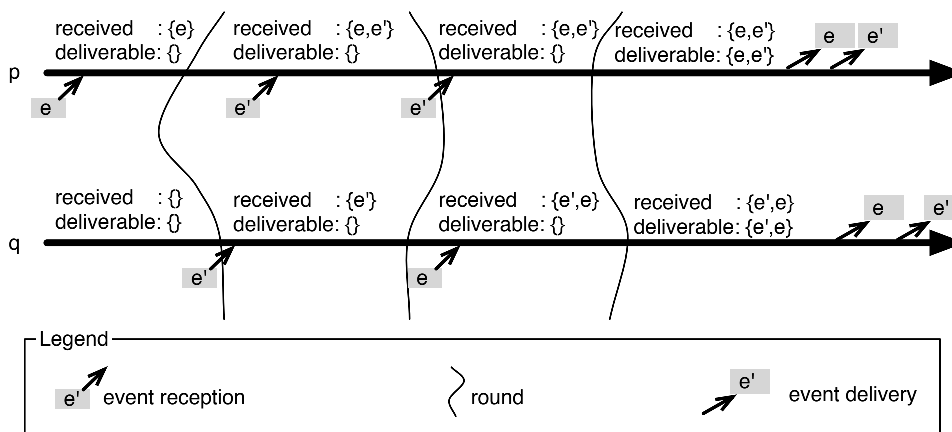


Figure 5.3: Totally ordered event delivery.

Most work on epidemic dissemination protocols focused only on reliability (Birman et al. 1999; Koldehofe 2002; Eugster et al. 2003b), that is, in satisfying the (probabilistic) agreement property.

Of particular interest to us is the balls-and-bins model, which offers a quantifiable probability of an event reaching all processes (Koldehofe 2002). In (Koldehofe 2002) processes are abstracted as bins, and balls represent a set of events. It is then shown that by throwing $c \cdot n \cdot lg(n)$ balls uniformly at random, with n the system size and $c > 1$ a constant, every bin will contain at least one ball w.h.p.

A protocol that creates $c \cdot n \cdot lg(n)$ balls is said to be *balls-and-bins compliant* and can be implemented by an epidemic algorithm as follows (Koldehofe 2002). In periodic rounds (not necessarily synchronized) each process sends balls to other K processes chosen uniformly at random. Each ball contains the events received during the last round that have been retransmitted less than TTL times. The fanout K is given by $\lceil 2 \cdot e \cdot ln(n)/ln(ln(n)) \rceil$ and the TTL by $\lceil lg(n) \rceil$ for a system of size n.

We now provide the intuition on how we can provide totally ordered event delivery on top of a ball-and-bins protocol. The protocol proceeds in asynchronous rounds and, for now, let us assume that processes have access to a global clock that provides totally ordered timestamps used to tag every broadcast event. We say that an event is *deliverable* when it has been received by all processes w.h.p. This information is conveyed to processes through an *isDeliverable* oracle derived from the TTL value of the balls-and-bins model. It follows that, for any two deliverable events e and e', a process p knows that every other correct process has received them w.h.p.

As events carry totally ordered timestamps, p can deterministically sort both events and deliver them in correct order to the application (Figure 5.3). Extending this observation to every process and pair of events, we are able to totally order all events without requiring any sort of global coordination among processes.

Should the deliverability oracle be perfectly accurate, the sequence of events delivered at each process would be exactly the same. However, as it is probabilistic, our algorithm will ensure the total order property at the expense of holes in the sequence of delivered events at some (in practice very few) processes.

In the remainder of this section we describe the algorithm in detail and discuss how our assumptions can be implemented in practice.

5.2.4 Detailed description

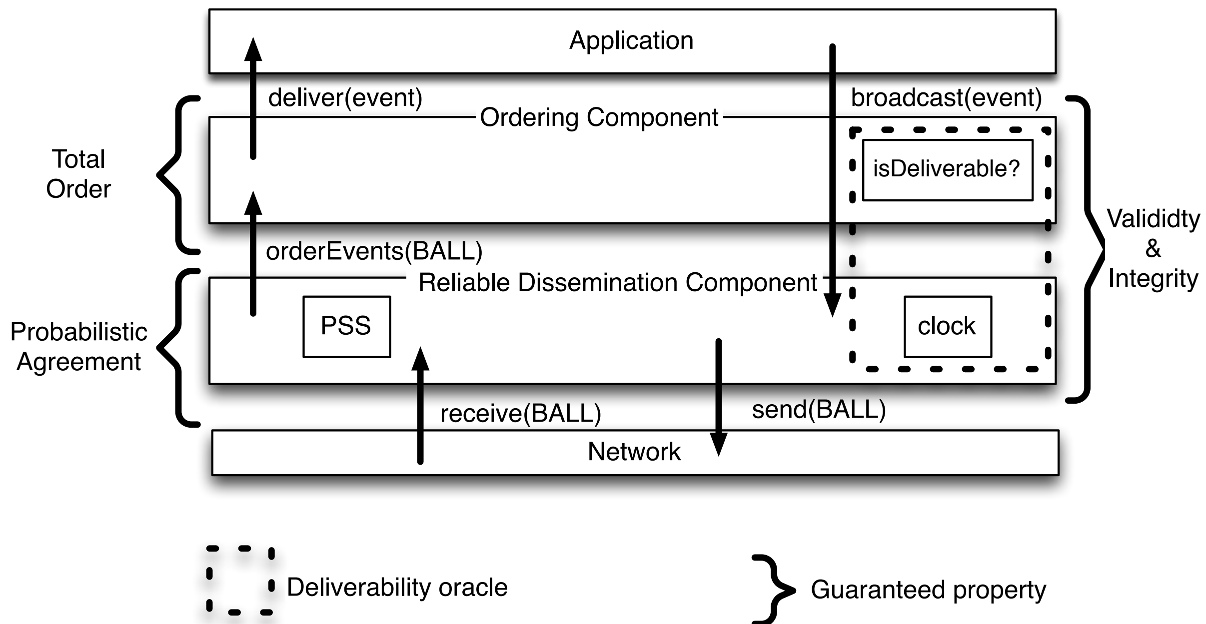


Figure 5.4: EpTO architecture.

The EpTO algorithm is composed of two parts: a balls-and-bins compliant dissemination component responsible for satisfying the agreement property and an ordering component responsible for fulfilling the total order property as depicted in Figure 5.4. The remaining properties of validity and integrity are ensured by the two components in tandem. We discuss how these properties are satisfied in Section 5.2.6. The dissemination component handles the reception and retransmission of events. Received events are passed to the ordering component which orders and delivers them to the application.

The dissemination component of the algorithm is depicted in Algorithm 4 for a process p. It proceeds in asynchronous rounds by periodically executing the task in lines 20 to 28. The algorithm assumes the existence of a PSS responsible for keeping p's view variable (line 2) up-to-date with a stream of at least K correct process, the fanout, for each round (Section 2.2.2). TTL is a constant holding the number of times each event needs to be relayed throughout its dissemination. The $nextBall$ set collects the events to be sent in the next round by p.

This component consists of three procedures executed atomically: the event broadcast primitive, the event receive callback and the periodic relaying task.

Algorithm 4: EpTO - reliable dissemination component (process p)

```
1  initially
2      view ← ...;               // system parameter: set of uniformly random correct peers
3      K ← ...;                                          // system parameter: fanout
4      TTL ← ...;           // system parameter: nb times events need to be relayed
5      nextBall ← ∅ ;                  // set of events to be relayed in the next round
6  procedure broadcast(event)
7      event.ts ← getClock()
8      event.ttl ← 0
9      event.sourceId ← p.id
10     nextBall ← nextBall ∪ (event.id, event)
11 upon receive BALL(ball)
12     foreach event ∈ ball do
13         if event.ttl < TTL then
14             if event.id ∈ nextBall then
15                 if nextBall[event.id].ttl < event.ttl then
16                     nextBall[event.id].ttl ← event.ttl ;              // update TTL
17             else
18                 nextBall ← nextBall ∪ (event.id, event)
19         updateClock(event.ts) ;                   // only needed with logical time
20 every δ
21     foreach event ∈ nextBall do
22         event.ttl ← event.ttl + 1
23     if nextBall != ∅ then
24         peers ← Random (view,K)
25         foreach q ∈ peers do
26             send BALL(nextBall) to q
27     orderEvents(nextBall)
28     nextBall ← ∅
```

When p broadcasts an event (lines 6–10), the event is timestamped with p's current clock, its *ttl* is set to zero and it is added to the *nextBall* to be relayed in the next round. Upon reception of a ball (lines 11–19), events with $ttl < TTL$ are added to *nextBall* for further relaying. When a received event is already in *nextBall*, we keep the one with the largest *ttl* to avoid excessive retransmissions. Finally, the process clock is updated (this will only become relevant later in Section 5.2.5).

The periodic relaying task is executed every δ time units (lines 20–28). Process p first updates the *ttl* of each event in *nextBall* and then sends it to K processes randomly chosen from its *view*. Next, it calls the procedure *orderEvents* of the ordering component (Algorithm 5) and afterwards resets the *nexBall*.

The ordering component is depicted in Algorithm 5. Procedure *orderEvents* is called every round (line 27 of Algorithm 4) and its goal is to deliver events to the application (Algorithm 5, line 30). To do so, each process p maintains

Algorithm 5: EpTO - ordering component (process p)

```
1  initially
2      received ← ∅ ;                                   // map of received but not delivered events
3      delivered ← ∅ ;                                  // set of delivered events
4      lastDeliveredTimestamp ← 0 ;                     // maximum timestamp of delivered events
5  procedure orderEvents(ball)
       // update TTL of received events
6      foreach event ∈ received do
7          received[event.id].ttl ← received[event.id].ttl + 1
       // update set of received events with events on the ball
8      foreach event ∈ ball do
9          if event.id ∉ delivered ∧ event.ts < lastDeliveredTimestamp then
10             if event.id ∈ received then
11                 if received[event.id].ttl < event.ttl then
12                     received[event.id].ttl ← event.ttl
13             else
14                 received ← received + (event.id, event)
       // collect deliverable events and determine smallest timestamp of non deliverable events
15     minQueuedTimestamp ← ∞
16     deliverableEvents ← ∅
17     foreach event ∈ received do
18         if isDeliverable(event) then
19             deliverableEvents ← deliverableEvents ∪ event
20         else if minQueuedTimestamp > event.ts then
21             minQueuedTimestamp ← event.ts
22     foreach event ∈ deliverableEvents do
23         if event.ts > minQueuedTimestamp then
               // ignore deliverable events with timestamp smaller than all received events
24             deliverableEvents ← deliverableEvents \ event
25         else
               // event can be delivered, remove from received events
26             received ← received − (event.id, event)
27     foreach event ∈ deliverableEvents | sorted by (timestamp, sourceId) do
28         delivered ← delivered ∪ event
29         lastDeliveredTimestamp ← event.ts
30         deliver(event) ;                              // deliver event to the application
```

a *received* map of $(id, event)$ pairs with all known but not yet delivered events and a *delivered* set with all the events already delivered to the application. The main task of this procedure is to move events from the *received* to the *delivered* set, preserving the total order of the events' timestamps. This is done in several steps as follows.

We start by incrementing the *ttl* of all events previously received (lines 6–7) as the result of entering a new round. Then, in lines 8 to 14, the events received in parameter *ball* are processed. For each event, if it has been already delivered or its timestamp is greater than the timestamp of the last event delivered

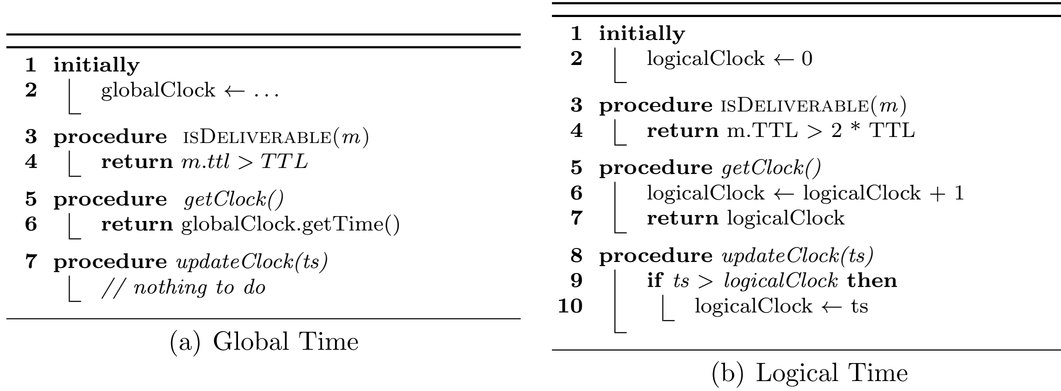


Figure 5.5: Stability oracles.

($lastDeliveredTimestamp$) then it is discarded. Delivering such an event in the former case would violate integrity because the delivery of a duplicate event, and in the latter case incur in a violation of total order. Otherwise, the event is added to *received* or, if previously there, its *ttl* value set to the largest of both occurrences. Note that here the event's *ttl* is not used anymore for dissemination but just for deliverability detection purposes as we explain below when discussing the *isDeliverable* oracle (Section 5.2.5).

The next step (lines 15–26) builds the set of events to be delivered in the current round ($deliverableEvents$). In a nutshell, an event e becomes deliverable if it is deemed so by the *isDeliverable* oracle and its timestamp is smaller than any non deliverable event in the *received* set. Lines 15 to 21 collect the deliverable events in the $deliverableEvents$ set and calculate the minimum timestamp ($minQueuedTimestamp$) of all the events that cannot yet be delivered. Next, lines 22 to 26 purge from the $deliverableEvents$ set all events whose timestamp is greater than $minQueuedTimestamp$ because otherwise total order would be violated. The remaining events can be effectively delivered, and thus are removed from the *received* set. Finally (lines 27–30), the events in $deliverableEvents$ are delivered to the application following timestamp ordering.

5.2.5 Deliverability oracle and logical time

In this section we discuss the implementation of the deliverability oracle. The concept of deliverability builds on the notion of event stability. An event is said to be stable once it has been received by all correct processes w.h.p. This is

actually the fulfillment of the probabilistic agreement property and, as shown in (Pittel 1987; Koldehofe 2002), in a balls-and-bins compliant protocol happens after the event is disseminated TTL times. In fact, the dissemination of events actually corresponds to *aging* the events until they become stable. Whereas for reaching agreement processes do not need to become aware of when an event becomes stable, in our algorithm this is essential to determine when the event can be delivered while preserving total order. To do so, our algorithm not only periodically ages events by disseminating the events in *nextBall* at each round (recall that these are the events received since the last dissemination occurred) in lines 21 to 26 in Algorithm 4, but also mimics at the same time the aging of events it has received and await to be delivered. This is done by incrementing their *ttl* in lines 6 and 7 of Algorithm 5. Therefore, because the events can be totally ordered, by assumption, thanks to the global clock, the deliverability oracle comes down to comparing the event's *ttl* with the system's TTL parameter, Figure 5.5(a).

Intuitively, for adequately mimicking the aging of events we need to approximate that aging with the dissemination periods. This approximation is controlled by the parameter δ which specifies how often the algorithm's round should be executed. For performance, δ should be a good estimate of the end-to-end communication delay to allow events to become stable. Note that because EpTO is asynchronous, setting a wrong δ only impacts performance not correctness. As a matter of fact, the safety of the algorithm (total order and integrity) is deterministic and always preserved as well as its liveness. Validity is also deterministic and always ensured while agreement is probabilistic, meaning not all messages might be delivered. In practice, this means that in the worst case, the sequence of events delivered at processes may contain holes. We informally discuss how these properties are satisfied in Section 5.2.6 and evaluate the impact of δ in EpTO's performance in Section 5.3.

Logical clock We now consider the use of logical time releasing the assumption of a global clock. We use a scalar logical clock implemented in a standard way: the local clock is incremented whenever an event is broadcasted and received with procedures *getClock*() and *updateClock*(*ts*) of Figure 5.5(b), respectively. By disambiguating concurrent events using the process identifiers, we are able to still totally order all events.

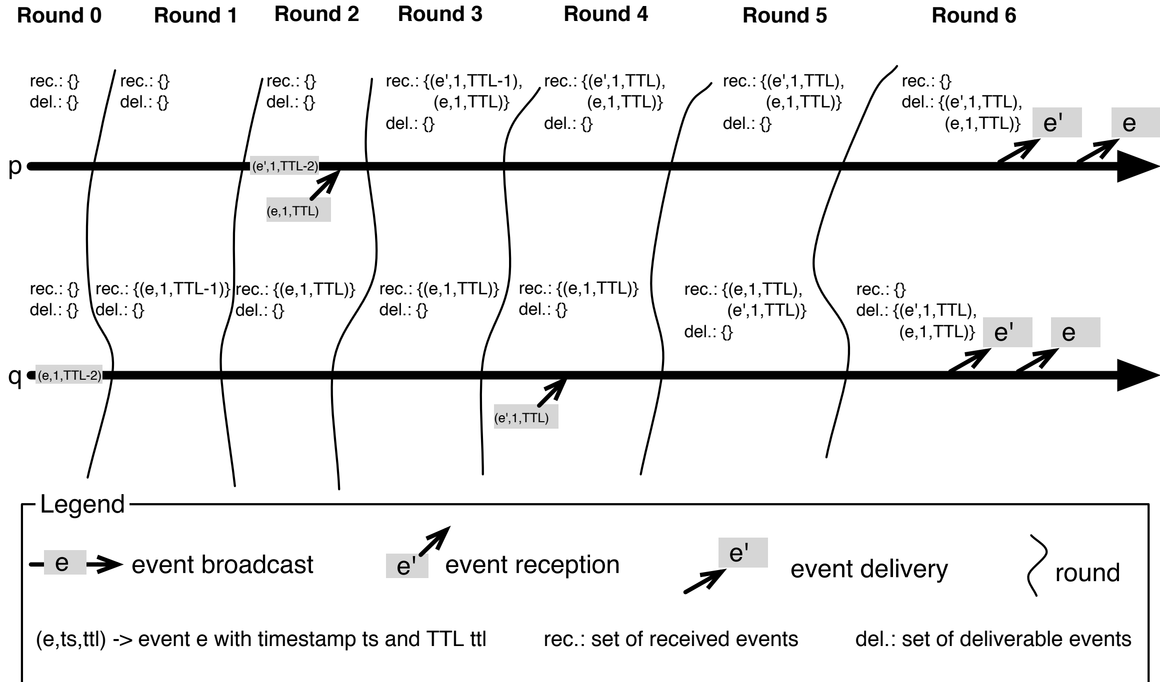


Figure 5.6: Sample run with a logical clock. Note that rounds are labeled just for presentation purposes, EpTO does not require round synchronization or labeling.

Because event stability depends on real time (rounds of δ units of time) but now timestamps are logical (and therefore oblivious to the passing of time), the event stability achieved at TTL rounds may not be sufficient to allow ordered delivery of concurrent events. To understand the phenomenon, consider the following example depicted in Figure 5.6 with processes p and q. Further assume that in this example, the TTL is two, i.e. events are stable after two rounds, processes initial logical clock is set to one and $p.id$ precedes $q.id$. Note that rounds are labeled just for the sake of explanation, EpTO does not require round synchronization or labeling. Process q broadcasts $(e, 1)$ (event e with timestamp 1) at round zero. Process p receives event e in round two but just before the reception, it broadcasts event $(e', 1)$. Because p broadcasts e' before receiving e, the timestamp associated with e' still does not take into account the timestamp of e, and thus both events have the timestamp set to one. Simultaneously e is deemed stable at q because TTL rounds have elapsed. Now, if our only criteria was event stability, q would correctly deliver e. However, by doing so q would no longer be able to deliver e' as it would violate total order. This is because $p.id$ precedes $q.id$, and thus e' precedes e, which could well be the order in which p will deliver

both events. The result will be an unnecessary hole in the sequence of delivered events at q. To overcome this issue events can only be considered deliverable when another TTL rounds have passed, i.e. $2 * TTL$. The reason for this is that after $2 * TTL$, any event received will already take into account the timestamp of events broadcasted $2 * TTL$ (TTL for the event to become stable w.h.p. plus another TTL for any concurrent event to also become stable w.h.p.) before, thus enabling gapless delivery w.h.p. In the example depicted in Figure 5.6 the result is that both processes deliver e' and e in that order at round six (i.e. $2*TTL = 4$ rounds after event e' was broadcasted).

5.2.6 Properties satisfiability

In this section we informally discuss how EpTO satisfies the Validity, Integrity, Agreement and Total Order properties. As highlighted in Figure 5.4, agreement is achieved by the dissemination component (Algorithm 4) while Total Order is obtained by the ordering component (Algorithm 5). Validity and Integrity are ensured by the two components in tandem. In the following we enumerate a set of lemmas that will support our propositions. For simplicity, and following the name of the data structures used in the alorithm listings, we call *received* to the map of received events and *delivered* to the set of delivered events (Algorithm 5, lines 2 and 3, respectively).

Lemma 1 *The timestamp of an event e broadcasted by a process p is greater than the timestamp of any event in* received.

The timestamp of event e is assigned when broadcasting the event with the call to procedure *getClock*() (Algorithm 4, line 7). The mechanism to update the clock is different whether we have a global or logical clock. We consider both cases next. Remember that, by assumption, procedures execute atomically. Furthermore, consider *lastReceivedTimestamp* to be the smallest timestamp in *received.*
Global clock case. From the algorithm, events are exclusively created, and the timestamp assigned, in the $BROADCAST$ procedure (Algorithm 4, lines 6–10). Moreover, events can be added to *received* only through the call to *orderEvents* (Algorithm 4, line 27). Because procedures execute atomically, the last execution of *orderEvents* happened before, in real time, than the call to $BROADCAST$. Moreover, because no events are created in the *orderEvents* procedure, any event

in *received* has been necessarly created before (in real time) its invocation. It follows that the timestamp of e, as given by the call to *getClock*(), is greater than that of any event in *received*.
Logical clock case. Process p can put in *received* events broadcasted by p itself or by other processes. Because process p is correct, the timestamp p assigns to the events it broadcasts monotonically increases (Algorithm 5.5(b)), and thus it is not possible to have an event e' created by p after event e with a timestamp smaller than e. Therefore, the remaining situation to consider is the timestamp of events created by other processes. These events are first received in the call to $BALL$ (Algorithm 4, lines 11–19), placed in the *nextBall* and their timestamp used to increase p's logical clock (Algorithm 4, line 19). It follows that the timestamp of any received event is used to update p's logical clock, and thus any event broadcasted by p will have a timestamp greater than $lastReceivedTimestamp$.

Lemma 2 *Only events in* received *are added to* delivered *(and delivered to the application).*

Directly from the algorithm. Only events belonging to the $deliverableEvents$ set are added to *delivered* (Algorithm 5, lines 27–28), and all events from $deliverableEvents$ come from *received* (Algorithm 5, lines 17–19).

Lemma 3 *An event e broadcasted by a correct process p is added to p's* received.

When a correct process p broadcasts an event e (Algorithm 4, lines 6–10), it is placed in the *nextBall* to be relayed on the next round. Correct processes will eventually execute the next round (Algorithm 4, lines 20–28), and thus will call the $orderEvents$ procedure (Algorithm 4, line 27) of the ordering component. Provided the following conditions hold, then event e will be added to the *received* (Algorithm 5, line 14). These conditions are: (i) event e does not belong to *delivered* (Algorithm 5, line 9), and (ii) the timestamp of e is greater than the timestamp of the last delivered event (Algorithm 5, line 9), and (iii) event e is not already in *received*.

Condition (i): event e does not belong to *delivered*. From the algorithm, it follows that *delivered* is only manipulated on the *orderEvents* procedure (Algorithm 5, line 28). Because this is the first invocation of *orderEvents* since event

e was broadcasted, then clearly event e cannot belong to *delivered.* Condition (i) is therefore satisfied.

Condition (ii): the timestamp of e is greater than the timestamp of the last delivered event. From Lemma 2 only events in *received* are added to *delivered.* Therefore, no event in *delivered* has a timestamp greater than an event in *received.* Moreover, from Lemma 1, the timestamp of any broadcast event e is greater than the timestamp of any event in *received.* It follows that condition (ii) holds.

Condition (iii): event e is not already in *received.* If e is already on *received*, then there is nothing left to do. Otherwise, the condition is true and e is added to *received* (Algorithm 5, lines 10–14). It follows that condition (iii) holds.

From conditions (i), (ii) and (iii) it follows that an event e broadcasted by a correct process p is added to p's *received.*

Lemma 4 *An event e in* received *eventually becomes stable.*

An event is said to be stable when its *ttl* is greater than TTL, and thus the event is known by all correct processes w.h.p. (Section 5.2.5). From the algorithm, the *ttl* of each event in *received* monotonically increases every round in two possible ways: by one unit (Algorithm 5, lines 6–7) or by one unit or more if the received ball has the same event with a larger *ttl* (Algorithm 5, lines 10–12). Moreover, TTL is a finite quantity so eventually the *ttl* of each event will be greater than TTL, and thus the event will be stable.

Lemma 5 *An event e in* received *eventually becomes the event with the smallest timestamp.*

In every round, *received* might be modifed in two different ways: by adding new events received in a ball (Algorithm 5, lines 8–14) and removing deliverable events (Algorithm 5, lines 22–26). Thus, one needs to consider two conditions: condition (i): events removed from *received* have timestamps smaller than e and condition (ii) events added to *received* eventually have timestamps greater then e.

Condition (i): events removed from *received* have timestamps smaller than e. From Lemma 4 events eventually become stable, and from Lemma 6 the stable event with the smallest timestamp is removed from *received.* Thus, events with timestamp smaller than e get removed from *received* before e does.

Condition (ii): events added to *received* eventually have timestamps greater then e. From the algorithm events can be added to *received* either if they are broadcasted by the process p itself (Algorithm 4, lines 6–10) or if they are received in a ball (Algorithm 4, lines 11–19). From Lemma 1 any event broadcast by p has a timestamp greater than any event in *received* so we only need to be concerned with events broadcasted by other processes. Faulty processes eventually crash so we only need to be concerned with correct processes. The timestamp at any correct process q also increases monotonically and, in the case of logical clocks, it will eventually take into account the timestamp of event e (see Section 5.2.5). This implies that eventually any event broadcasted by a process q will have a timestamp larger than e.

From condition (i) and condition(ii) it follows that event e will eventually become the event with the smallest timestamp in *received*.

Lemma 6 *The stable event e with the smallest timestamp in* received *is added to* delivered *and removed from* received*.*

Any event belonging to the *deliverableEvents* set is added, in timestamp order, to *delivered* (Algorithm 5, lines 27–30). Thus, we need to consider how the *deliverableEvents* set is built. The *deliverableEvents* set is initially populated with the events that are deliverable according to the *isDeliverable* oracle (Algorithm 5, lines 18–19). Remember that an event e is said to be deliverable if and only if e is stable and no event created afterwards can have a timestamp smaller than e w.h.p. (Section 5.2.5). For all events in *received* that are not deliverable, we determine the smallest timestamp *minQueuedTimestamp* of such events (Algorithm 5, lines 20–21). Next, all events in the *deliverableEvents* set whose timestamp is greater than *minQueuedTimestamp* (Algorithm 5, lines 22–24). It follows that *deliverableEvents* ends up containing all the deliverable, and thus stable, events with timestamps smaller than all non deliverable events. These events are removed from *received* (Algorithm 5, lines 25–26) and added to *delivered* in timestamp order (Algorithm 5, lines 27–30).

Lemma 7 *An event e is added to* delivered *(and delivered to the application) at most once.*

Note that *received* is a map data structure with the event identifier as the key. This means that at any given time, *received* does not hold the same event

more than once. Similarly, *delivered* is a set, and thus it cannot contain the same event more than once.

An event can be added to *received* only if it is not in *delivered* (Algorithm 5, lines 8–14). Besides, any event added to *delivered* is removed from *received* (Algorithm 5, lines 23–28). This means that gradually events are moved from *received* to *delivered*. Therefore, because by assumption procedures are executed atomically, it is not possible to add an event to *received* that is already in *delivered*. Moreover, from Lemma 2 only events in *received* can be added to *delivered*. It follows that events are added to *delivered* and delivered to the application at most once.

Lemma 8 *The dissemination of an event e terminates after TTL retransmissions.*

When a new event e is broadcasted (Algorithm 4 lines 6–10), its *ttl* is set to zero and it is added to the *nextBall*. On the next round (Algorithm 4 lines 20–26), the *ttl* of all events contained in the *nextBall* is increased by one and the ball is sent to K peers. Upon reception of a *ball* (Algorithm 4 lines 11–19), the events in that ball are included in p's *nextBall* if and only if the event's *ttl* is smaller than TTL. It follows that only events with *ttl* smaller than TTL will be relayed, thus ensuring termination of events' dissemination in TTL rounds.

Lemma 9 *Events are disseminated to K processes chosen uniformly at random.*

From the algorithm (Algorithm 4, lines 24–26) it is clear that only events in the *nextBall* are transmitted. As observed before, an event can be put into the *nextBall* set either if it is broadcasted by a process p (Algorithm 4 lines 6–10), and hence its *ttl* is set to zero, or by Lemma 8 if it was received in a ball and its *ttl* is smaller than TTL.

These events contained in *nextBall*, are sent to K processes, a protocol parameter. The target processes are chosen by the RandomPick function (Algorithm 4, line 26) which selects from the *view* variable, K process uniformly at random (Section 2.4). By assumption, the *view* variable is managed by a PSS and contains uniformly random samples from the network (Section 2.2.2). Because a uniformly random selection from a uniformly random variable preseves the uniformly random properties the target processes are chosen uniformly at random.

Lemma 10 *No process p adds an event e' to* delivered *after an event e if the timestamp of e' is smaller than the timestamp of e.*

From Lemma 2 only events in *received* are delivered to the application and added to *delivered*. If both events e and e' are in *received* simultaneously, then from Lemma 4, Lemma 5 and Lemma 6, e' will be delivered before e and thus this lemma holds. Additionally, from the same three lemmas it follows that eventually all events in *received* will be added to *delivered*. Thus, the problem arises only when e is already on *delivered* and e' is not yet in *received*. Suppose this is the case. When and if e' is received, provided its dissemination was not terminated (Lemma 8), i.e its *ttl* was not expired, then e' is added to the *nextBall* (Algorithm 4, lines 12–18) to be disseminated and passed to the *orderEvents* procedure in the next round (Algorithm 4, lines 23–27). Note that if the dissemination of event e' already terminated, then e' will be simply not added to the *nextBall*, and thus it is not eligible for delivery in the next round. It follows that we need to consider only events whose dissemination was not yet terminated. For e' to be added to *received* in the *orderEvents* procedure, three conditions are required: condition (i) e' cannot be in *delivered*, condition (ii) the timestamp of e' needs to be greater than the timestamp of the last delivered event, *lastDeliveredTimestamp*, and condition (iii) e' cannot be in *received* (Algorithm 5, lines 9–10). By assumption, e' is neither in *received* nor in *delivered*, so conditions (i) and (iii) fail, respectively. Also by assumption, event e is already in delivered, and thus by the algorithm the *lastDeliveredTimestamp* is at least set to the timestamp of e. But, by assumption the timestamp of event e is greater than the timestamp of e'. It follows that event e' cannot be added to the *received* and, by Lemma 2, cannot be added to *delivered*.

Propositions We now discuss how, based on the previous lemmas, EpTO satisfies the Validity, Integrity, Probabilistic Agreement and Total Order properties.

Proposition 1 (Integrity) EpTO *satifies the Integrity property.*

With respect to Integrity we need to consider two sufficient and necessary conditions: integrity-a) only broadcasted events are delivered and integrity-b) no event is delivered more than once.
Integrity-a: only broadcasted events are delivered. By Lemma 2 only events

in *received* can be *delivered.* Events can be added to *received* only if they are contained in a ball passed to the *orderEvents* procedure (Algorithm 5). There are only two ways events can become part of a ball, either they are broadcasted locally by the process p (Algorithm 4, lines 6–10) or they are received in a ball (Algorithm 4, lines 11–19). Balls are received from other processes, and contain events they have broadcasted or received. Besides, by assumption, channels do not duplicate or create spurious messages (Section2.1). It follows that only broadcasted events are delivered.
Integrity-b: no event is delivered more than once follows directly from Lemma 2 and Lemma 7.

Proposition 2 (Validity) EpTO *satifies the Validity property.*

By Lemma 3 an event e broadcasted by a correct process p is added to p's *received.* An event e in p's *received* eventually becomes stable by Lemma 4 and with the smallest timestamp by Lemma 5. Moreover, the stable event with the smallest timestamp is added to *delivered* by Lemma 6. It follows that if a correct process broadcasts an event e, then it eventually delivers e.

Proposition 3 (Probabilistic Agreement) EpTO *satifies the Probabilistic Agreement property.*

The satisfiability of Agreement comes directly from the balls-and-bins model. Under the balls-and-bins model it is enough to retransmit a ball (set of one or more events) $c \cdot n \cdot lg(n)$ times uniformly at random for that ball to fall into all bins (processs) w.h.p. Specifically, an epidemic protocol is balls-and-bins compliant if the following conditions hold: agreement-a) it retransmits $c \cdot n \cdot lg(n)$ copies of each event, agreement-b) to processes chosen uniformly at random and agreement-c) terminates after TTL rounds (Koldehofe 2002).
Agreement-b and agreement-c follow directly from Lemma 9 and Lemma 8, respectively.

Finally, condition agreement-a requires that $c \cdot n \cdot lg(n)$ copies of each event are created and disseminated. From agreement-b and agreement-c it is clear that each process creates, at each round, K copies of each event whose *ttl* was not expired. It follows that for any $K \geq 1$ it is enough to set the TTL to $c \cdot n \cdot lg(n)$, for any $c \geq 1$ to create enough events to satisfy condition agreement-a.

Note however that the actual bound on the TTL is much lower, as given by (Koldehofe 2002). As a matter of fact, to satisfy Agreement w.h.p. the TTL needs to be $\lceil lg(n) \rceil$ and the fanout $K = \lceil 2 \cdot e \cdot ln(n)/ln(ln(n)) \rceil$ for a system of size n. From the combination of agreement-a, agreement-b and agreement-c it follows that EpTO satisfies Agreement w.h.p.

Proposition 4 (Total Order) EpTO *satifies the Total Order property.*

Regardless of the clock used, all events broadcasted can be deterministically sorted by timestamp and then by broadcast identifier. Suppose two correct processes p and q and that both have events e and e' in *delivered*. Moreover, assume the timestamp of event e is smaller than the timestamp of e'. We proceed by contradiction. Suppose that total order was violated, i.e. the order by which p and q have added events e and e' to *delivered* is different. For this to happen, one process delivered e before e' (the correct order) and the other process delivered e' before e (the wrong order). Without loss of generality, let us assume that p is the process which delivered the events in the wrong order. This means that process p added e to *delivered* after adding e', despite the timestamp of e' being greater than the timestamp of e. However, from Lemma 10 this is a contradiction. It follows that EpTO satisfies the Total Order property.

Regarding performance.

From the discussion above, we saw that EpTO satisfies the properties of a Total Order algorithm specification (Défago et al. 2004) subject to the Probabilistic Agreement. Of particular interest is the interplay between the Probabilistic Agreement and Total Order. As a matter of fact this is the key for scalability in the number of events and processes (Section 5.3) and allows us to discard events, and thus *blame* Agreement, whenever Total Order is endangered. Naturally, we want to avoid such discards as much as possible. Next, we discuss the conditions required to avoid forced event discards. For the sake of this discussion, let us assume that forced discards were not performed, and thus violations in the Total Order property could occur. It follows that the only way for events to be put in *delivered* in different order by two processes p and q is if the $deliverableEvents$ set at p differs from the one at q when the $orderEvents$ is executed independently at each process. Note that this does not impose any synchronization

between processes, it just requires that whenever the *deliverableEvents* set is populated (Algorithm 5 lines 15–26) the set of events it contains are the same. From Lemma 6 the stable event with the smallest timestamp in *received* is added to *delivered* and from Lemma 2 events eventually become stable. Events become stabler by increasing their *ttl* which can happen in two ways: when they are received in a ball (which corresponds to the dissemination in the balls-and-bins model) and due to the local stabilization procedure we discussed in Section 5.2.4. In both cases, the pace at which the *ttl* of events is increased is controlled by the round period δ, which indicates how often processes should retransmit unstable events and how often the local stabilization should be run. Next, we discuss two scenarios where a poorly chosen δ was a negative impact on the number of forced discards. Note that because the system is asynchronous, the correctness properties discussed above always hold irrespective of the performance of the system. We consider two scenarios: scenario-a) processes cycle faster than the network delay and scenario-b) processes drift considerably and place bounds on each.

For scenario-a assume a system with two processes p and q with δ set to $\frac{networkdelay}{TTL+1}$. Furthermore, suppose p and q broadcast events e and e' at the same time, respectively. From the algorithm, both processes will locally detect its own events as stable after $TTL \cdot \delta$ time units (Lemma 4). Because $\frac{networkdelay}{TTL+1} \cdot TTL < networkdelay$, then each process will locally stabilize its event before receiving the other process's event. It follows that from Lemma 5 and Lemma 6 both p and q will deliver e and e', respectively. In the next round p and q receive e' and e, respectively. Now, if $e.ts < e'.ts$, p can deliver e' but q cannot from Lemma 10. Strictly, this avoids a total order violation but creates an unnecessary hole in the sequence of delivered events. To prevent this, δ needs to be at least $\frac{networkdelay}{TTL-1}$ to give broadcasters enough time to receive event from other processes before delivering their own. However, in practice, and to the best of our knowledge, using round periods so small will be a waste of network resources. In fact, this issue is not discussed in most epidemic dissemination protocols we are aware of (Birman et al. 1999; Koldehofe 2002; Eugster et al. 2003b). Our simulations on the pathological cases (Figure 5.10(a)) confirm that this bound can indeed be just on the order of the median network delay (it's enough to reach half the nodes) but below that we are prone to get holes in the delivery sequence.

The scenario-b case can happen if a process's round period drifts too much

from the configured δ and can be divided in two sub-cases: future-drift (running too quickly) and past-drift (running too slowly). The future-drift is just a particular case of scenario-a, and thus has the same effects. The past-drift has the opposite effect because events broadcasted by the process itself will take longer to be injected into the network. This means that other processes might receive this process's events too late and have to drop them, creating unnecessary holes. For both cases of the drift is bound $\pm\delta$, which in practice means that the process may execute the next round immediately or execute it twice as slower than expected. These results are also confirmed by our simulations on the pathological cases (Figure 5.10(b)).

5.3 Evaluation

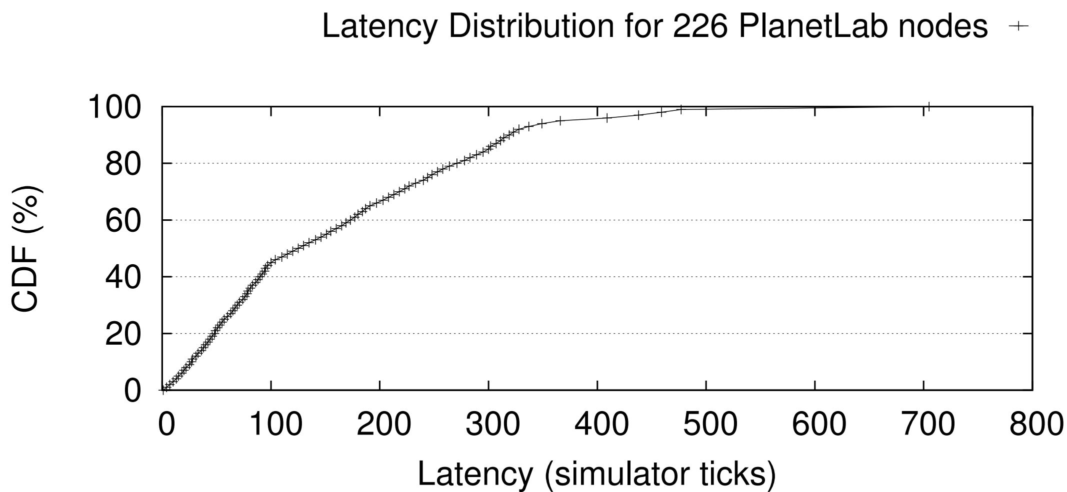
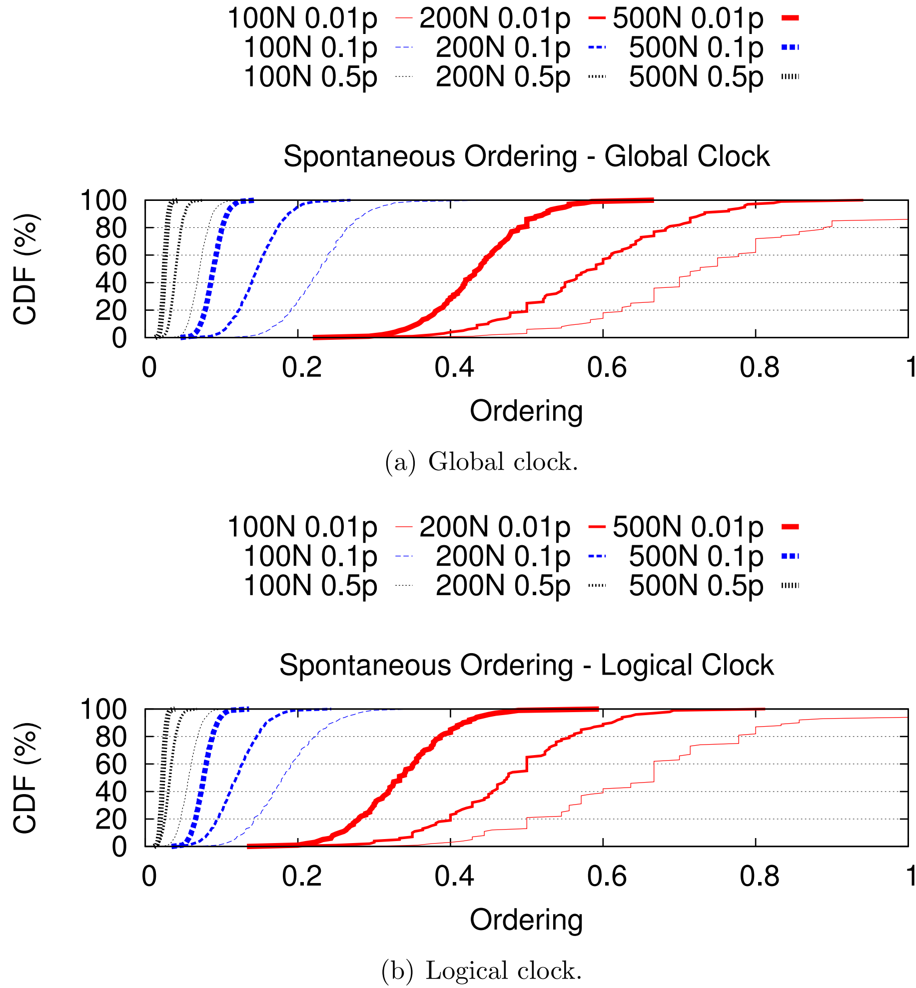


Figure 5.7: Latency distribution used in experiments (obtained from PlanetLab).

In this section we evaluate EpTO's performance under several scenarios and analyze in which pathological conditions and misconfigurations the performance degrades. The experiments are conducted in the same simulator used to evaluate StaN (Section 3.3) with the following configurations. Processes execute at time $now() + \delta \pm processDrift$, balls sent are delivered at processes at time $now() + networkLatency$ and processes may be added/removed from the system at a rate $churnRate$.

In all the experiments below, $networkLatency$ is a random sample from an



(a) Global clock.

(b) Logical clock.

Figure 5.8: Spontaneous order with a global and a logical clock for varying system sizes and publication rates r.

empirical distribution obtained by pinging among PlanetLab nodes as depicted in Figure 5.7. The mean is ≈ 157 with standard deviation ≈ 119 and the 5th , 50th and 95th percentiles are 15, 125 and 366 simulator ticks, respectively.

Experiments are run with 100, 200, and 500 processes for event publication rates, r, of 0.01, 0.1, and 0.5. An event publication rate of r means that each processes has, in each round, a probability r of broadcasting a new event. The main task of the processes executes every $\delta = 125$ ticks (the median network latency) with a $processDrift$ of 0.1, meaning the execution period is randomly distributed in the interval [113..137] (i.e. 125 ± 12). All presented results are the average of 20 runs.

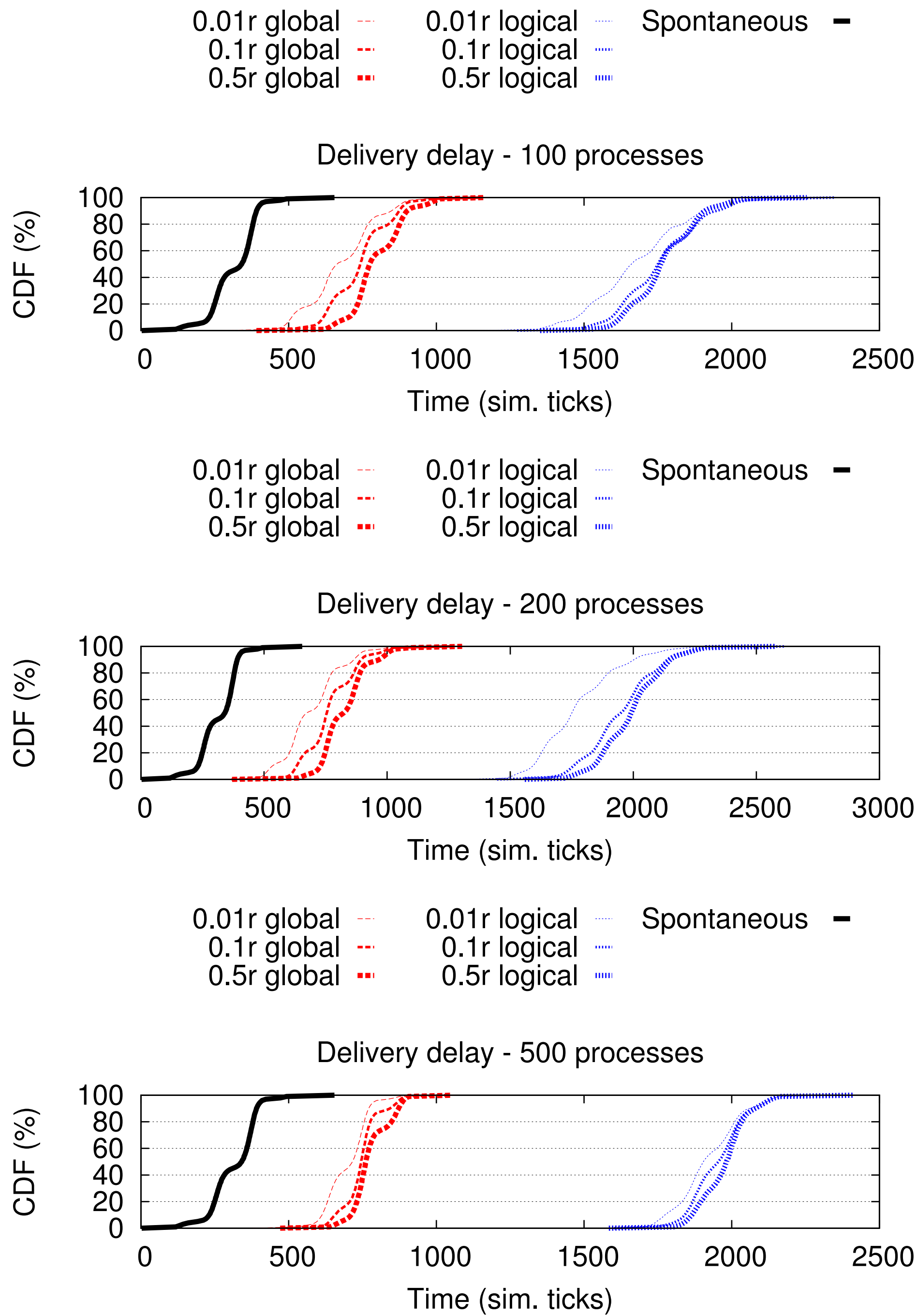
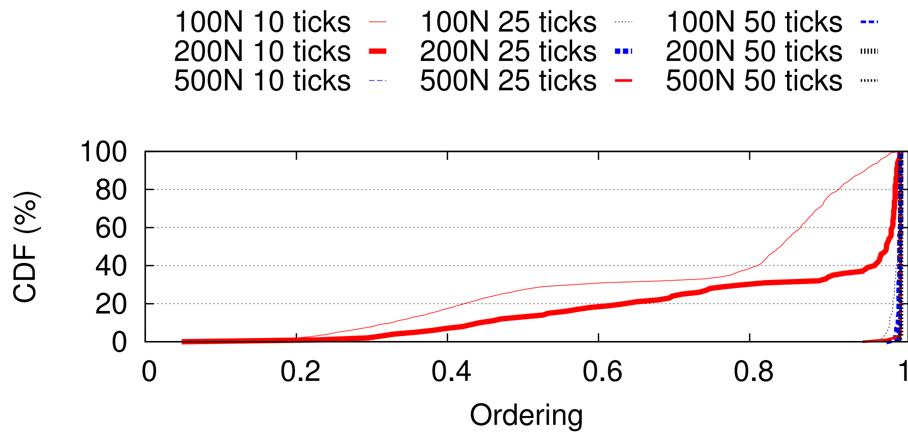
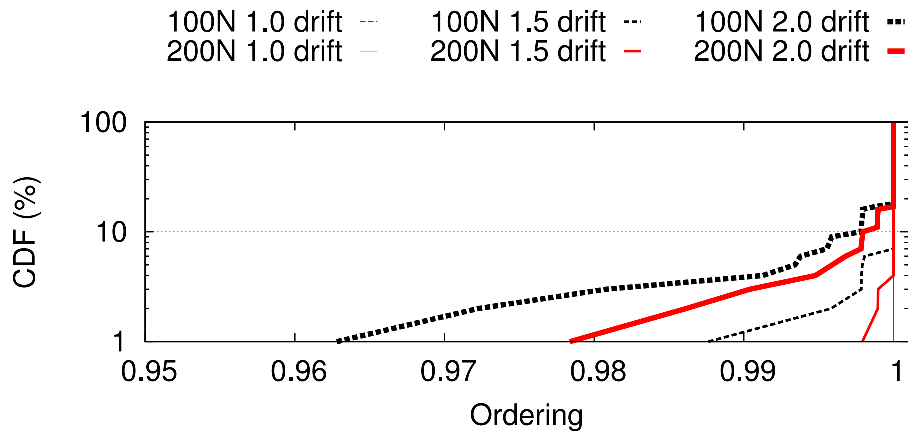


Figure 5.9: Delivery delay for 100, 200 and 500 processes for varying publication rates r and clock types.

We first measure the spontaneous order of the system when using only an epidemic dissemination protocol. This corresponds to delivering new events to



(a) Misconfigured task period δ.



(b) Task execution period drift for a baseline $\delta = 125$.

Figure 5.10: Pathological disorder situations related to the task execution period δ for a publication rate $r = 0.5$.

the application right after they are received for the first time (Algorithm 4, line 11). Results are presented in Figure 5.8. The ordering is measured by computing the common sequences between the events delivered at processes, and the final event sequence as given by deterministically sorting all events by timestamp and process identifier. As expected, the ordering becomes worse (i.e. more disordered) with increasing number of processes and events, but it is much more affected by the latter due to multiple events being broadcasted concurrently. For broadcast ratios higher than 0.1 no process is able to correctly deliver more than half of its events, which makes spontaneous ordering in epidemic dissemination protocols unusable. In fact, the inherent randomness of an epidemic dissemination clearly

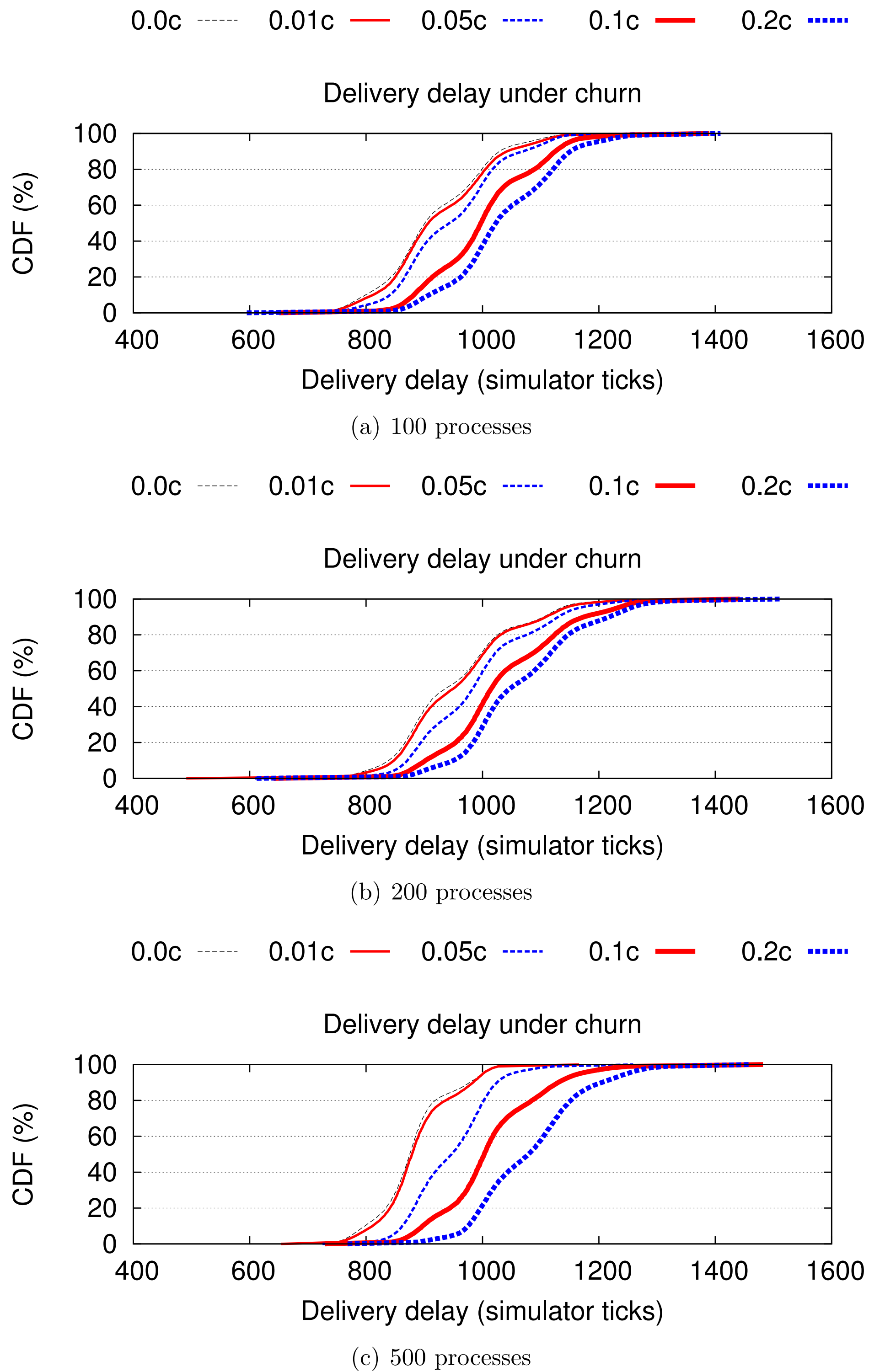
goes against any spontaneous ordering that might arise from the natural network order as exploited in other works (Sousa et al. 2002; Pedone and Schiper 2003). When using EpTO in the same scenario all processes deliver all events in the same order, either with a global or logical clock.

We next focus on the delivery delay of EpTO, which measures the time spent, in simulator ticks, between the broadcast of an event and its delivery to the application. This indicates the cost of reaching order when compared to a typical epidemic dissemination protocol. Results are shown in Figure 5.9. The first observation is that the delay for achieving order is two to four times that of an unordered epidemic protocol depending on whether we can access a global clock. Given the much stronger abstraction, a total order algorithm offers to the application (note that all events were delivered by all processes in the same order), EpTO offers a reasonable compromise. Moreover, EpTO also scales well both in the number of nodes and events, as increasing those fivefold only results in about a twofold increase in the delay.

In the following experiment we study two pathological cases where performance degrades. The first is when the period of execution δ of the algorithm's main task (Algorithm 4) is too short compared to the network delay. This means that events are locally stabilized (Algorithm 5, lines 6–7) before having the chance to reach all nodes, and thus preclude agreement. We experimented with several values for the task period corresponding to different percentiles of the network delay (Figure 5.7). Results are presented in Figure 5.10(a). As expected, when the periodic task executes too fast the performance degrades because processes end up with holes in the sequence of delivered events. Still, EpTO is surprisingly resilient to this misconfiguration and just starts to degrade importantly for values below a third of the median network delay as hinted in Section 5.2.6. The other pathological case is when process drift is very large, i.e. processes may execute significantly slower or faster than the configured period. Despite being a different cause, the practical effect is the same as before: events might be prematurely stabilized, thus precluding the delivery of "earlier" events to the application. We also experimented with several values for the allowed $processDrift$, as shown in Figure 5.10(b). Note that for drifts larger than one, we prevent processes from executing the next round "in the past" and instead execute it immediately. Remarkably, even with a drift as large as the task execution period (1.0 drift lines

in the figure), which means the next execution of the task can happen almost immediately or take twice the configured value, 99% of the processes are able to deliver all events in the same order, i.e. without holes. For larger values the performance starts to degrade quickly as discussed in Section 5.2.6.

Finally, we study the behavior of EpTO under churn by observing the evolution of the delivery delay. To this end, we subject half of the system to a given churn rate by removing *churnRate* percent nodes uniformly at random and adding *churnRate* percent nodes to the system every $\delta = 125$ simulator ticks. We then measure the ordering on processes not subject to churn. Results are presented in Figure 5.11. As one can observe, the impact of churn on the delivery delay is almost unnoticeable with churn rates up to 10%. Moreover, for churn rates up to 20% all events were delivered to all nodes in the same order. This highlights the robustness of EpTO for dynamic environments, which stems essentially from its epidemic nature.



(a) 100 processes

(b) 200 processes

(c) 500 processes

Figure 5.11: Impact of churn on the delivery delay with a global clock and publication rate $r = 0.5$.

5.4 Related Work

There is a vast amount of work on deterministic total order algorithms, much of which has been analyzed and categorized in (Défago et al. 2004). To the best of our knowledge, the earliest idea on a probabilistic total order algorithm is briefly discussed in (Hayden and Birman 1996). The algorithm uses an epidemic dissemination protocol and, similarly to EpTO, waits for messages to become stable before delivering them. However, its dissemination protocol has weaker guarantees than a balls-and-bins compliant dissemination protocol, and thus it is more likely to violate agreement. The detection of stability is based on the real time expected for an event to reach all nodes, unlike EpTO rounds, and it is thus very sensitive to fluctuations in the network delay and process drift. Because of that, it is also restricted to scenarios with access to a global clock.

Some optimistic total order algorithms also rely on the notion of real time required for an event to reach all nodes by exploiting the spontaneous network order (Sousa et al. 2002; Pedone and Schiper 2003). The goal is to decrease the delivery latency of deterministic protocols and allow applications to process events optimistically. However, such protocols still require a deterministic delivery of the final order, and thus are subject to the same scalability constraints of traditional protocols. Besides, they are sensitive to network fluctuations, and thus prone to mistakes in the optimistic delivery. The possibility of doing mistakes requires optimistic protocols to define corrective mechanisms that fix those mistakes and define the final deterministic order. Mistakes in EpTO are avoided by *blaming* agreement meaning that events which would result in an order violation are dropped at the expense of the final deterministic order of optimistic protocols.

The PABCast protocol proposed in (Felber and Pedone 2002) proceeds in asynchronous rounds where processes can either broadcast an event or vote for other processes' events. Processes communicate through gossip and exchange the set of events and its voters. A round terminates when processes collect $n - f$ votes (n being the system size and f the number of faulty processes) and deterministically deliver all events. PABCast provides probabilistic safety and liveness properties, whereas EpTO provides deterministic safety (integrity and total order are always preserved) and probabilistic liveness (validity is preserved and agreement is achieved w.h.p). The basic version of PABCast only allows for

processes to either broadcast a single event or place a vote for an event of another process. This can be overcome with several extensions to the protocols but, as the authors point out, the number of concurrent broadcasted makes the protocol more prone to out of order deliveries.

5.5 Discussion

The ordering of events is one of the most fundamental and well studied problems in distributed systems (Lamport 1978; Défago et al. 2004). Until recently, the focus of research was on the construction of primitives with strong guarantees and sound theoretical basis. However, the impressive growth of distributed systems in terms of scale, started to expose the practical weaknesses of these approaches, namely poor scalability, quickly degrading behavior under churn and increasing latency. These issues, amplified by the scale of modern systems, led researchers to devise alternative formulations with weaker yet quantifiable guarantees, such as eventual consistency (Vogels 2009) or epidemic dissemination protocols themselves (Demers et al. 1987; Birman et al. 1999).

Part of the limitations of classical approaches stem, in our opinion, from the reliance on the deterministic behavior of the agreement property. As a matter of fact, the cost of a deterministic reliable multicast primitive - on which one can build the agreement property - is one of the reasons that led to the emergence of probabilistic dissemination mechanism and, in particular, to epidemic protocols (Demers et al. 1987; Birman et al. 1999). For instance, optimistic total order protocols generally leverage a degraded form of agreement to optimistically deliver events to the application, and thus overcome the latency constraints of a deterministic reliable multicast (Sousa et al. 2002; Saito and Shapiro 2005). In a second step, optimistic protocols still use a deterministic reliable multicast primitive to deliver the final order to the application and, if necessary, issue corrective deliveries to fix any mistakes done in the optimistic phase.

With EpTO, we explicitly rely on the use of a probabilistic reliable multicast primitive, and thus on probabilistic agreement. This means that in the absence of deterministic agreement, while total order is always ensured, the reliable delivery of events is probabilistic, and thus subject to holes in the sequence of delivered events. Still, the evaluation conducted shows that the impact of ensuring total

order in event reliability (i.e. preventing order violations) is just observed in the most extreme scenarios (Figure 5.10). As a matter of fact, because agreement is ensured with high probability, in normal scenarios the guarantees offered by EpTO are close to those of a deterministic algorithm. Actually, EpTO provides deterministic integrity and total order, and thus safety is deterministic. Validity is also deterministic and agreement is ensured w.h.p., thus liveness is ensured with high probability. The probabilistic nature of agreement, which sidesteps coordination, allows EpTO to scale to a large number of processes and events. The performance of the algorithm in terms of delivery delay can be fine tuned by adjusting the period δ of execution without impacting safety.

In EpTO, because agreement is probabilistic, one is allowed to drop events whenever its delivery would incur in an order violation. Alternatively, one can consider the delivery of corrective deliveries to fix mistakes as done in optimistic protocols. Note, however, that the absence of a final order in EpTO makes these corrective deliveries substantially different from the ones in optimistic protocols. As a matter of fact, in the latter, a corrective delivery as given by the final order is definitive, and thus enables the application to proceed accordingly. On the other hand, in EpTO one does not have a final order and as such it is not possible to issue a corrective delivery and inform the application that it is final. This is actually close to the notion of unconscious eventual consistency (Baldoni et al. 2006), where processes might receive corrective deliveries but are not aware (i.e. they are unconscious) if the delivery order they possess is definitive. Studying the suitability of EpTO to such a programing model is an interesting research avenue that we plan to pursue.

In complement to the unconscious programing model discussed above, one can also consider directly exposing to the application the probabilistic nature of agreement. In fact, from the balls-and-bins model which underlies the dissemination guarantees, one knows that after TTL dissemination rounds a given event is stable, i.e. it is known by all processes w.h.p. It is possible to go a step further and expose the notion of stability to the application by associating each known but not yet delivered event with the probability of being stable (and deliverable). Therefore, the application could peek into this list and decide, for each event, if the associated probabilities of stability and deliverability are acceptable and consume them accordingly. As before, we plan to formalize and develop this model

in future work.

Chapter 6

Conclusions

In this dissertation we focused on three fundamental problems in distributed systems. While the problems we target have been solved before using deterministic algorithms, existing state of the art solutions barely address, to the best of our knowledge, these same problems in very large scales. This stems from the scale of the system itself, the cost, for instance in terms of message complexity, of maintaining certain system properties, the outright adverse dynamics of the environment where churn and failures are the norm rather than the exception, and the impossibility of centralized knowledge and management.

The harsh environment and uncertain nature of very large scale distributed systems preclude stringent approaches based on rigid knowledge about the system, and thus forfeit the deterministic outcomes produced by classical approaches. The alternative considered in this dissertation - epidemic algorithms - are perfectly suited to these harsh and uncertain environments essentially due to their randomized nature which is able to produce outcomes otherwise unattainable in a deterministic environment (Ben-Or 1983).

Equipped with the basic philosophy and concepts of epidemic algorithms, we proceeded to formulate the research question leading to this dissertation:

> What key weaknesses preclude epidemic algorithms from being broadly applied to a wider range of very large scale scenarios?

Due to its message passing nature, research on epidemic algorithms usually focus on devising new message exchange patterns, or improving existing ones, to construct a service or address an application specific need. The loosely coupled

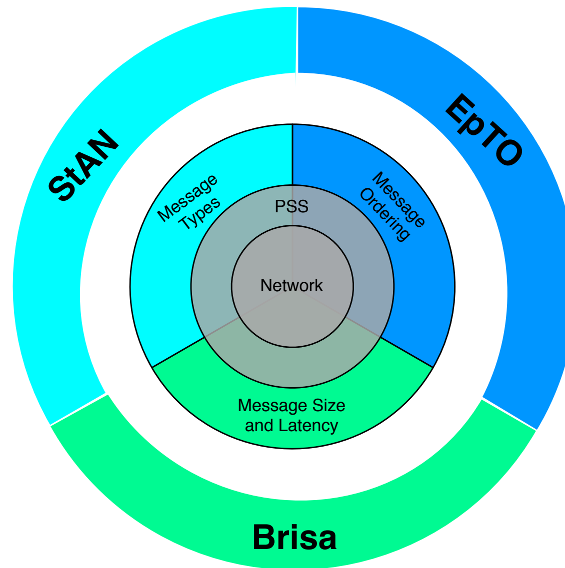


Figure 6.1: Placement of each proposed algorithm in the problem space.

nature of epidemic algorithms make them particularly suited to the publish-subscribe model and in particular to its topic-based variant (Eugster et al. 2003a). Existing algorithms, while scalable in the number of processes and messages, deal poorly with multiple topics and thus multiple message types categorized into those topics. Moreover, the distinguishing robustness of protocols comes at the expense of multiple redundant message transmissions (Demers et al. 1987). This makes epidemic protocols poorly suited to the dissemination of larger contents restricting their use mostly to control data. Finally, many distributed applications require messages to be ordered in a specific way, and in particular in total order. With a few notable exceptions (Hayden and Birman 1996; Felber and Pedone 2002), there is little research, to the best of our knowledge, in epidemic algorithms offering total order guarantees.

These three gaps led us to formulate three different research questions and propose algorithms to address them, as depicted in Figure 6.1.

Next, we will briefly discuss the major research questions introduced in the

beginning of this dissertation. However, instead of detailing the particular merits of each approach, as has already been done in their respective chapters, we will elaborate instead on the underlying concepts and principles that led to their design in the first place:

1. How can we deal with different message types and what is the impact on management overhead?
   This problem has been address by StaN, a scalable topic-based publish-subscribe algorithm described in Chapter 3. StaN takes advantage of the interest overlap of subscribers to reduce the management overhead associated with participating in multiple topics. The major novelty of StaN is, in our opinion, the acknowledgement that one cannot construct, at the same time, an overlay organization that scales in the number of topics by explicitly exploiting subscriber interests, and possesses the properties suitable for epidemic dissemination. We solve this conundrum through a weight metric completely unrelated to subscriptions overlap and thus able to sidestep their inherent clustering as the major cause of overlay degradation. The reduction in management overhead is thus not a cause of subscription correlation, as in other approaches, but its indirect consequence. As the optimization criteria is completely oblivious to subscriptions, StaN presents itself as a surprisingly effective mechanism to preserve the privacy and safety of subscribers. We intend to exploit these capabilities in future research.

2. How can we deal with large message sizes and what is the impact on bandwidth and latency?
   This problem has been addressed by Brisa, an efficient and reliable data dissemination algorithm described in Chapter 4. Brisa builds efficient dissemination structures, such as trees, backed up by a robust unstructured overlay that is able to quickly fix disruptions in the structure and thus minimize service disruptions. The major observation underlying Brisa is the recognition that what makes epidemic algorithms robust is the possibility, at any given instant, of receiving redundant messages, not the actual message transmission. With this in mind, Brisa disables the transmission of messages on most paths following some application specific criteria, but in such a way that all processes are reachable and connectivity can be promptly restored. The combination of efficiency and robustness, along

with the ability to quickly restore connectivity, makes Brisa an appealing solution for epidemic live streaming video dissemination.

3. How can we deal with message ordering, and in particular total order?
This problem has been addressed by EpTO, an epidemic total order dissemination algorithm described in Chapter 5. EpTO disseminates events following a balls-and-bins approach and when event stability is detected - i.e. when an event has reached all processes w.h.p. - delivers the event to the application in total order. The major novelty supporting EpTO is the conversion of a deterministic property hard to achieve in a large dynamic distributed setting - agreement -, into a probabilistic one that can be estimated from local knowledge only, based solely on the termination properties of the underlying balls-and-bins dissemination algorithm. This enables EpTO to offer a total order primitive, subject to probabilistic agreement, able to scale to a large number of processes and events.

Each one of the proposed algorithms adequately solves, in our opinion, the stated problems. Notwithstanding the particular research avenues and open challenges already discussed for each of the proposed algorithms, we believe that their combination and composition brings interesting possibilities. We discuss these combinations in the next section.

6.1 Future work

Building on the main problems specified in this dissertation, and on the proposed solutions, it is possible to derive a new set of problems by combining the basic ones as depicted in Figure 6.2. In the following, we briefly discuss these possibilities and point toward future research directions.

Message Types and Ordering. This is the problem of offering total order with messages of different types, i.e. belonging to multiple topics. At first sight, this could be addressed by the combination of StaN and EpTO. One can consider the simpler case of ordering of events just inside a single topic or a more complex scenario where ordering is required to encompass multiple topics. In the former, one might want to apply EpTO directly into an overlay (topic) managed

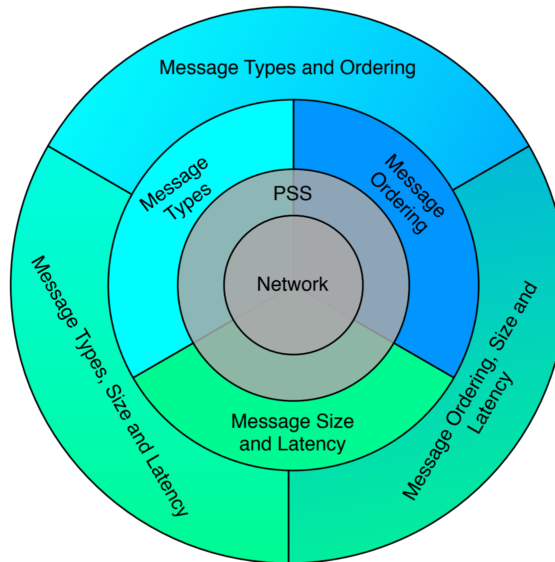


Figure 6.2: Possible problem combinations and challenges.

by StaN. However, the overlays managed by StaN are, by the very nature of the algorithm, reactive. This limits the randomness of the neighbors of a process which might negatively affect the dissemination guarantees of the balls-and-bins model, and thus probabilistic agreement. Studying whether this endangers probabilistic agreement of if there are better ways of ensuring probabilistic agreement in this scenario is thus an open question. The latter is clearly related to total order multicast to multiple groups (Guerraoui and Schiper 1997). Clarifying the differences between each approach and researching on whether one can improve existing state of the art based on the proposals presented here is also an interesting perspective.

Message Ordering, Size and Latency. This is the problem of totally ordering messages with large payloads and/or strict latency requirements. It could be addressed by a combination of EpTO and Brisa but it is not as straightforward as the previous problem. This is because due to the large size of the payloads, typically there are few sources, and more commonly just one, and thus ordering

can be done at the senders which eliminates the need for EpTO. Moreover, because large payloads are usually media, such as video or music, the usefulness of being able to totally order such content is not clear. Therefore, this research path is, in our opinion, the least interesting.

Message Size, Latency and Types. This is the problem of disseminating large message payloads over one or more topics. Oppositely to the previous problem, one can clearly see the usefulness of such primitive, for instance to disseminate media content together with news feeds in a topic-based publish-subscribe system. We can address this problem by a combination of Brisa and StaN. This combination is actually quite natural because the reactive nature of the overlay manage by StaN is clearly adequate to Brisa's requirements. However, some research questions remain. For instance, while it is straightforward to use Brisa in a single StaN topic to disseminate large payloads, studying how one can combine multiple topics with Brisa's mutliple trees becomes quite challenging and interesting. Moreover, the disclosure of message crossposting to Brisa also opens the door to more efficient topic-based publish-subscribe systems.

Bibliography

L.A. Adamic and B.A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3 (1):143–150, 2002. - **Cited** on page 38.

Akamai Technologies. Akamai. http://www.akamai.com, 2013. - **Cited** on page 6.

S. Baehni, P.T. Eugster, and R. Guerraoui. Data-aware multicast. In *Dependable Systems and Networks*, 2004. - **Cited** on pages 30 and 56.

Norman Bailey. *The Mathematical Theory of Infectious Diseases and its Applications.* Hafner Press, second edi edition, 1975. - **Cited** on pages 3 and 20.

R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni. TERA: topic-based event routing for peer-to-peer architectures. In *International conference on Distributed event-based systems*, 2007a. - **Cited** on pages 5, 26, 30 and 56.

Roberto Baldoni, Rachid Guerraoui, R Levy, V Quéma, and Sara Tucci Piergiovanni. Unconscious eventual consistency with gossips. *Stabilization, safety, and security of distributed systems*, 2006. URL `http://www.springerlink.com/index/m022w817t4326083.pdf`. - **Cited** on page 129.

Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. Tera: topic-based event routing for peer-to-peer architectures. In *Proceedings of the International Conference on Distributed Event-based Systems*, DEBS, pages 2–13, New York, NY, USA, 2007b. ACM. - **Cited** on page 92.

Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert.

StreamHub: A Massively Parallel Architecture for High-Performance Content-Based Publish/Subscribe. In *Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS '13*, page 63, New York, New York, USA, June 2013. ACM Press. ISBN 9781450317580. doi: 10.1145/2488222.2488260. URL `http://dl.acm.org/citation.cfm?id=2488222.2488260`. - **Cited** on page 25.

Michael Ben-Or. Another advantage of free choice (Extended Abstract). In *Proceedings of the second annual ACM symposium on Principles of distributed computing - PODC '83*, pages 27–30, New York, New York, USA, August 1983. ACM Press. ISBN 0897911105. doi: 10.1145/800221.806707. URL `http://dl.acm.org/citation.cfm?id=800221.806707`. - **Cited** on pages 3 and 131.

R Bhagwan, S Savage, and G Voelker. Understanding availability. In *Proc. of IPTPS: international workshop on Peer-to-Peer Systems*, February 2003a. - **Cited** on page 49.

Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Peer-to-Peer Systems II*, Lecture Notes in Computer Science, pages 256–267. Springer Berlin / Heidelberg, 2003b. - **Cited** on page 97.

Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/312203.312207. - **Cited** on pages 2, 3, 20, 26, 61, 62, 99, 100, 103, 119 and 128.

Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970. - **Cited** on page 70.

Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, New York, USA, July 2000. ACM Press. ISBN 1581131836. doi: 10.1145/343477.343502. URL `http://dl.acm.org/citation.cfm?id=343477.343502`. - **Cited** on pages 2 and 99.

Nuno A. Carvalho, José Pereira, Rui Oliveira, and Luís Rodrigues. Emergent Structure in Unstructured Epidemic Multicast. In *Proceedings of the 37th*

Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2855-4. doi: http://dx.doi.org/10.1109/DSN.2007.40. - **Cited** on pages 3, 21, 22 and 100.

Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, pages 219–227, New York, New York, USA, July 2000. ACM Press. ISBN 1581131836. doi: 10.1145/343477.343622. URL `http://dl.acm.org/citation.cfm?id=343477.343622`. - **Cited** on page 4.

M. Castro, P. Druschel, A.M. Kermarrec, and A.I.T. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002. - **Cited** on pages 4, 16, 19, 55, 61 and 91.

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP, pages 298–313, New York, NY, USA, 2003a. ACM. - **Cited** on pages 61, 63, 74, 76, 84, 92 and 96.

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP, pages 298–313, New York, NY, USA, 2003b. ACM. - **Cited** on page 7.

Miguel Castro, Michael Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An Evaluation of Scalable Application-Level Multicast Built Using Peer-to-Peer Overlays. In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1510–1520, 2003c. - **Cited** on page 20.

R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *International Conference on Parallel and Distributed Computing*, 2005. - **Cited** on page 27.

Chen Chen, Hans-Arno Jacobsen, and Roman Vitenberg. Divide and Conquer Algorithms for Publish/Subscribe Overlay Design. In *Int. Conference on Distributed Computing Systems*. IEEE, 2010. ISBN 978-1-4244-7261-1. doi: 10.1109/ICDCS.2010.87. - **Cited** on page 57.

Chen Chen, Roman Vitenberg, and Hans-Arno Jacobsen. Scaling Construction of Low Fan-out Overlays for Topic-Based Publish/Subscribe Systems. In *International Conference on Distributed Computing Systems*. IEEE, 2011. ISBN 978-1-61284-384-1. doi: 10.1109/ICDCS.2011.68. - **Cited** on page 57.

G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *Principles of Distributed Computing*, 2007a. - **Cited** on pages 5, 26, 27, 31 and 57.

G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *International Conference on Distributed Event-Based Systems*, 2007b. - **Cited** on pages 5, 26, 27, 30, 51, 56 and 59.

Y. Chu, S.G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, 20:1456–1471, 2002. - **Cited** on pages 4 and 61.

S. Cimmino, C. Marchetti, and R. Baldoni. A Guided Tour on Total Order Specifications. In *The Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS' 03)*, pages 187–187. IEEE, 2003. ISBN 0-1795-2054-5. doi: 10.1109/WORDS.2003.1267507. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1410962'escapeXml='false'/>. - **Cited** on page 8.

Bram Cohen. Incentives build robustness in bittorrent, 2003. - **Cited** on pages 4 and 6.

Bram Cohen. The bittorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, January 2008. - **Cited** on pages 4 and 6.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser,

Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Operating Systems Design and Implementation*, 2012. ISBN 978-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387905`. - **Cited** on page 101.

Alan Dcmers Dan, Carl Hauser, Wes Irish, John Larson, Scott Shenkcr, Howard Sturgis, Dan Swinehart, Doug Terry, Alan Demers, Dan Greene, and Scott Shenker. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987. ISBN 0-89791-239-4. doi: http://doi.acm.org/10.1145/41840.41841. - **Cited** on page 100.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205, October 2007a. ISSN 01635980. doi: 10.1145/1323293.1294281. URL `http://dl.acm.org/citation.cfm?id=1323293.1294281`. - **Cited** on page 4.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41:205–220, 2007b. - **Cited** on pages 3 and 62.

Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. In *ACM Computing surveys*, volume 36, 2004. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.3146`. - **Cited** on pages 4, 7, 8, 99, 102, 118, 127 and 128.

Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th ACM Symposium*

on Principles of distributed computing, PODC, pages 1–12, New York, NY, USA, 1987. ACM. - **Cited** on pages 2, 3, 20, 62, 65, 87, 128 and 132.

P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2), 2003a. ISSN 0360-0300. - **Cited** on pages 5, 25, 26 and 132.

Patrick Eugster, Rachid Guerraoui, Sidath Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, 2003b. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/945506.945507. - **Cited** on pages 3, 20, 21, 26, 30, 31, 61, 62, 100, 103 and 119.

Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. From Epidemics to Distributed Computing. *IEEE Computer*, 37(5): 60–67, May 2004. - **Cited** on pages 7, 17, 20, 28 and 31.

C. Feather. Network News Transfer Protocol (NNTP). RFC 3977 (Proposed Standard), October 2006. URL `http://www.ietf.org/rfc/rfc3977.txt`. Updated by RFC 6048. - **Cited** on page 21.

Zongming Fei and Mengkun Yang. A proactive tree recovery mechanism for resilient overlay multicast. *IEEE/ACM Transactions on Networking*, 15:173–186, 2007. - **Cited** on pages 93 and 94.

Pascal Felber and Fernando Pedone. Probabilistic Atomic Broadcast. In *International Symposium on Reliable Distributed Systems*, 2002. URL `http://portal.acm.org/citation.cfm?id=831138`. - **Cited** on pages 8, 127 and 132.

Mario Ferreira, Joao Leitao, and Luis Rodrigues. Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems*, SRDS, pages 293–302, New Delhi, India, 2010. IEEE Computer. - **Cited** on pages 94, 95 and 96.

Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/3149.214121. URL `http://doi.acm.org/10.1145/3149.214121`. - **Cited** on page 2.

S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997. ISSN 10636692. doi: 10.1109/90.650139. URL `http://dl.acm.org/citation.cfm?id=270856.270863`. - **Cited** on page 2.

P. Fraigniaud, P. Gauron, and M. Latapy. Combining the use of clustering and scale-free nature of exchanges into a simple and efficient P2P system. In *International Conference on Parallel and Distributed Computing*, 2005. - **Cited** on pages 28, 34 and 38.

D. Frey, R. Guerraoui, A.-M. Kermarrec, M. Monod, and V. Quema. Stretching gossip with live streaming. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, pages 259–264, Budapest, Hungary, 2009. IEEE Computer Society. - **Cited** on page 61.

Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Maxime Monod. Boosting Gossip for Live Streaming. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10. IEEE, August 2010. ISBN 978-1-4244-7140-9. doi: 10.1109/P2P.2010.5569962. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5569962`. - **Cited** on page 21.

Robert Gallager, Pierre Humblet, and Philip Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1), 1983. ISSN 0164-0925. URL `http://portal.acm.org/citation.cfm?doid=357195.357200`. - **Cited** on page 16.

Ayalvadi Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. In *Networked Group Communication*, Lecture Notes in Computer Science, pages 44–55. Springer Berlin / Heidelberg, 2001. - **Cited** on pages 3, 7, 17, 21, 28, 31 and 62.

Ayalvadi Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. HiScamp: self-organizing hierarchical membership protocol. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 133–139. ACM, 2002. - **Cited** on page 17.

John Gantz. The Expanding Digital Universe. Technical report, IDC White Paper - sponsored by EMC, 2007. URL `http://www.emc.com/collateral/analyst-reports/expanding-digital-idc-white-paper.pdf`. - **Cited** on pages 1, 61 and 96.

John Gantz. The Diverse and Exploding Digital Universe. Technical report, IDC White Paper - sponsored by EMC, 2008. URL `http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf`. - **Cited** on pages 1, 61 and 96.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002. ISSN 01635700. doi: 10.1145/564585.564601. URL `http://dl.acm.org/citation.cfm?id=564585.564601`. - **Cited** on pages 2 and 99.

Sarunas Girdzijauskas, Gregory Chockler, Ymir Vigfusson, Yoav Tock, and Roie Melamed. Magnet: practical subscription clustering for Internet-scale publish/subscribe. In *International Conference on Distributed Event-Based Systems*. ACM Press, 2010. ISBN 9781605589275. doi: 10.1145/1827418.1827456. - **Cited** on pages 5 and 55.

C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks: algorithms and evaluation. *Performance Evaluation - P2P Computing Systems*, 63(3), 2006. ISSN 0166-5316. - **Cited** on page 18.

R Guerraoui and A Schiper. Total order multicast to multiple groups. *Distributed Computing Systems, 1997. ...*, 1997. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.7078http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=603426`. - **Cited** on page 135.

R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001. ISSN 00985589. doi: 10.1109/32.895986. URL `http://dl.acm.org/citation.cfm?id=359555.359565`. - **Cited** on page 2.

Rachid Guerraoui, Rui Oliveira, and André Schiper. Stubborn communication channels. Technical report, Tech. Rep.98-278, Département d'Informatique,

École Polytechnique Fédérale de Lausanne, 1998. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.7698`. - **Cited** on page 14.

Indranil Gupta, R Van Renesse, and KP Birman. A probabilistically correct leader election protocol for large groups. In Maurice Herlihy, editor, *Distributed Computing*, volume 1914 of *Lecture Notes in Computer Science*, pages 89–103, Berlin, Heidelberg, March 2000. Springer Berlin Heidelberg. ISBN 978-3-540-41143-7. doi: 10.1007/3-540-40026-5. URL `http://www.springerlink.com/content/k8tfgwnq6fa590q2http://link.springer.com/chapter/10.1007/3-540-40026-5_6`. - **Cited** on page 3.

S. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the eDonkey network, and implications for the design of server-less file sharing systems. *ACM Eurosys*, 2006. - **Cited** on pages 28, 34 and 38.

Mark Hayden and Kenneth Birman. Probabilistic Broadcast. Technical Report TR96-1606, Cornell University, 1996. URL `http://dl.acm.org/citation.cfm?id=866882`. - **Cited** on pages 100, 127 and 132.

John Jannotti, David Gifford, Kirk Johnson, M. Frans Kaashoek, and James O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Usenix OSDI Symposium 2000*, pages 197–212, October 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.7544`. - **Cited** on page 16.

Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8–es, August 2007a. ISSN 07342071. doi: 10.1145/1275517.1275520. URL `http://portal.acm.org/citation.cfm?doid=1275517.1275520http://dl.acm.org/citation.cfm?id=1275517.1275520`. - **Cited** on pages 14 and 58.

Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25, 2007b. - **Cited** on pages 3, 17, 28, 30, 31, 62 and 64.

Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), aug 2007c. - **Cited** on pages 26, 27 and 46.

Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 53:2321–2339, 2009. - **Cited** on pages 27, 32, 92 and 97.

S. Jun and M. Ahamad. Feedex: collaborative exchange of news feeds. In *Int. Conference on World Wide Web*, 2006. - **Cited** on page 25.

R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Symposium on Foundations of Computer Science*. IEEE Computer Society, 2000. - **Cited** on page 20.

A-M. Kermarrec, L. Massoulié, and A. Ganesh. Probabilistic reliable dissemination in large-scale systems. *Transactions on Parallel and Distributed Systems*, 14, 2001. - **Cited** on pages 7, 17 and 42.

Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. NAT-resilient Gossip Peer Sampling. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 360–367. IEEE, June 2009. ISBN 978-0-7695-3659-0. doi: 10.1109/ICDCS.2009.44. URL `http://dl.acm.org/citation.cfm?id=1584339.1584601`. - **Cited** on page 14.

Boris Koldehofe. Simple gossiping with balls and bins. *International Conference on Principles of Distributed Systems*, 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.2404&rep=rep1&type=pdf`. - **Cited** on pages 100, 101, 103, 104, 109, 117, 118 and 119.

Boris Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22nd IEEE International Symposium on Reliable Distributed Systems*, SRDS, pages 76–85, Florence, Italy, 2003. IEEE Computer. doi: 10.1109/RELDIS.2003.1238057. - **Cited** on pages 35 and 72.

Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of*

the nineteenth ACM symposium on Operating systems principles, SOSP, pages 282–297, New York, NY, USA, 2003. ACM. - **Cited** on page 92.

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, April 2010. ISSN 01635980. doi: 10.1145/1773912.1773922. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1309162`. - **Cited** on page 4.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978. ISSN 00010782. doi: 10.1145/359545.359563. URL `http://dl.acm.org/citation.cfm?id=359545.359563`. - **Cited** on pages 7, 8, 99 and 128.

J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *International Conference on Dependable Systems and Networks (IEEE DSN)*, pages 419–428. IEEE Computer Society, 2007. - **Cited** on page 17.

João Leitão, Robbert van Renesse, and Luís Rodrigues. Balancing gossip exchanges in networks with firewalls. page 7. USENIX Association, April 2010. URL `http://dl.acm.org/citation.cfm?id=1863145.1863152`. - **Cited** on page 14.

Joao Leitão, José Pereira, and Luís Rodrigues. Epidemic Broadcast Trees. In *Proceedings of the 22nd IEEE International Symposium on Reliable Distributed Systems*, SRDS, pages 301–310, Beijing, China, 2007a. IEEE Computer. - **Cited** on page 94.

Joao Leitão, José Pereira, and Luís Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, pages 419–429, Edinburgh, Scotland, 2007b. IEEE Computer Society. - **Cited** on pages xvi, 3, 17, 62, 64, 66 and 73.

Lorenzo Leonini, Etienne Rivière, and Pascal Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Symposium on Networked Systems Design and Implementation*, NSDI, pages

185–198, Berkely, CA, USA, 2009. Usenix Association. - **Cited** on pages 10, 41 and 76.

Z. Li, G. Xie, and Z. Li. Towards reliable and efficient data dissemination in heterogeneous peer-to-peer systems. In *Proceedings of the 22th IEEE International Parallel and Distributed Processing Symposium*, IPDPS, pages 1–12, Miami, FL, USA, 2008. IEEE Computer Society. - **Cited** on page 93.

Zhenyu Li, Gaogang Xie, Kai Hwang, and Zhongcheng Li. Churn-resilient protocol for massive data dissemination in p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 22:1342–1349, 2011. - **Cited** on page 93.

Jin Liang, Steven Ko, Indranil Gupta, and Klara Nahrstedt. MON : On-demand Overlays for Distributed System Management. In *Proceedings of the 2nd conference on Real, Large Distributed Systems*, WORLDS, pages 13–18, Berkely, CA, USA, 2005. Usenix Association. - **Cited** on pages 61, 63 and 93.

Meng Lin and Keith Marzullo. Directional Gossip: Gossip in a Wide Area Network. In *Proceedings of Third European Dependable Computing Conference*, volume 1667 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 1999. URL `http://link.springer.de/link/service/series/0558/bibs/1667/16670364.htm`. - **Cited** on page 17.

H. Liu, V. Ramasubramanian, and E.G. Sirer. Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In *Internet Measurement Conference*, 2005. - **Cited** on pages 27 and 38.

Jiangchuan Liu and Ming Zhou. Tree-assisted gossiping for overlay video distribution. *Multimedia Tools and Applications*, 29:211–232, 2006. - **Cited** on pages 21, 63 and 87.

LiveJournal, Inc. `http://www.livejournal.com/stats.bml`, 2013. - **Cited** on page 37.

Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-pull peer-to-peer live streaming. In *Distributed Computing*, Lecture Notes in Computer Science, pages 388–402. Springer Berlin / Heidelberg, 2007. - **Cited** on page 7.

M. Luby. *Pseudorandomness and Cryptographic Applications.* Princeton University Press, 1994. ISBN 0691025460. - **Cited** on pages 31 and 32.

NA Lynch. *Distributed algorithms.* Morgan Kaufmann Publishers Inc., January 1996. ISBN 1558603484. URL http://dl.acm.org/citation.cfm?id=525656http://books.google.com/books?hl=en&lr=&id=2wsrLg-xBGgC&oi=fnd&pg=PP2&dq=Distributed+Algorithms&ots=G3sWtxG2zv&sig=xEP9uJhxBKrbg2mJMzyAPrSct7I. - **Cited** on pages 2 and 13.

L. Massoulié, A-M. Kermarrec, and A. Ganesh. Network awareness and failure resilience in self-organising overlay networks. In *Symposium on Reliable Distributed Systems*, 2003. - **Cited** on pages 17 and 27.

Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Principles of Distributed Computing*, 2006. - **Cited** on page 18.

Miguel Matos, Ana Nunes, Rui Oliveira, and José Pereira. Stan: Exploiting shared interests without disclosing them in gossip-based publish/subscribe. In *Proc. of IPTPS: international workshop on Peer-to-Peer Systems*, San Jose, CA, USA, 2010. - **Cited** on page 41.

M. McGlohon. *Structural Analysis of Large Networks: Observations and Applications.* PhD thesis, Carnegie Mello University, 2010. - **Cited** on pages 35 and 53.

Wang Mea, Li Baochun, Mea Wang, and Baochun Li. R2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming. *IEEE Journal on Selected Areas in Communications*, 25(9):1655–1666, December 2007. ISSN 0733-8716. doi: 10.1109/JSAC.2007.071205. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4395125. - **Cited** on page 97.

Roie Melamed and Idit Keidar. Araneola: A scalable reliable multicast system for dynamic environments. *Journal of Parallel and Distributed Computing*, 68 (12):1539–1560, 2008. - **Cited** on page 17.

Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Chord on demand. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, P2P, pages 87–94, Washington, DC, USA, 2005. IEEE Computer Society. - **Cited** on pages 3 and 62.

Anh Tuan Nguyen, Baochun Li, and Frank Eliassen. Chameleon: Adaptive Peer-to-Peer Streaming with Network Coding. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, March 2010. ISBN 978-1-4244-5836-3. doi: 10.1109/INFCOM.2010.5462032. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5462032. - **Cited** on pages 21 and 97.

A. Nunes, J. Marques, and J. Pereira. Seeds: The social internet feed caching and dissemination architecture. In *INForum Simpósio de Informática*, 2009. - **Cited** on page 25.

Melih Onus and Andréa W. Richa. Parameterized Maximum and Average Degree Approximation in Topic-Based Publish-Subscribe Overlay Network Design. In *International Conference on Distributed Computing Systems*. IEEE, 2010. ISBN 978-1-4244-7261-1. doi: 10.1109/ICDCS.2010.54. - **Cited** on page 57.

J. Patel, E. Rivière, I. Gupta, and A.-M. Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, 53(13):2304–2320, August 2009a. ISSN 13891286. - **Cited** on pages 26 and 37.

Jay A. Patel, Etienne Rivière, Indranil Gupta, and Anne-Marie 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, 2009b. - **Cited** on page 92.

Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1), 2003. ISSN 03043975. doi: 10.1016/S0304-3975(01)00397-8. URL http://dl.acm.org/citation.cfm?id=795635.795644. - **Cited** on pages 124 and 127.

J. Pereira, L. Rodrigues, R. Oliveira, and A.-M. Kermarrec. Neem: Network-friendly epidemic multicast. In *Symposium on Reliable Distributed Systems*, 2003. - **Cited** on pages 21 and 28.

Fabio Pianese, Diego Perino, JoaquÍn Keller, and Ernst Biersack. PULSE: An Adaptive, Incentive-Based, Unstructured P2P Live Streaming System. *IEEE Transactions on Multimedia*, 9(8):1645–1660, 2007. ISSN 15209210. doi: 10.1109/TMM.2007.907466. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4378428`. - **Cited** on page 22.

Boris Pittel. On Spreading a Rumor. *SIAM Journal on Applied Mathematics*, 47 (1), 1987. ISSN 0036-1399. doi: 10.1137/0147013. URL `http://dl.acm.org/citation.cfm?id=37387.37400`. - **Cited** on page 109.

PlanetLab. PlanetLab. http://www.planet-lab.org, 2013. - **Cited** on pages 41, 64 and 76.

Charles Plaxton, Rajmohan Rajaraman, and Andréa Richa. Accessing nearby copies of replicated objects in a distributed environment. *ACM Symposium on Parallel Algorithms and Architectures*, 1997. URL `http://portal.acm.org/citation.cfm?doid=258492.258523`. - **Cited** on page 15.

L. Querzoni. Interest clustering techniques for efficient event routing in large-scale settings. In *International Conference on Distributed Event-Based Systems*, 2008. - **Cited** on page 57.

Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Workshop on Networked Group Communication*, NGC '01. Springer-Verlag, 2001. - **Cited** on pages 16, 19 and 55.

Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, Middleware, pages 389–409, New York, Inc. New York, NY, USA, 2007. Springer-Verlag. - **Cited** on pages 3, 62 and 63.

Etienne Rivière and Spyros Voulgaris. *Gossip-Based Networking for Internet-Scale Distributed Systems*, volume 78 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

ISBN 978-3-642-20861-4. doi: 10.1007/978-3-642-20862-1. URL `http://www.springerlink.com/content/g472j742413p6457/`. - **Cited** on page 3.

Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, Lecture Notes in Computer Science, pages 329–350. Springer Berlin / Heidelberg, 2001. - **Cited** on pages 16, 91 and 92.

Laura S. Sabel and Keith Marzullo. Election Vs. Consensus in Asynchronous Systems. February 1995. URL `http://dl.acm.org/citation.cfm?id=866766`. - **Cited** on page 2.

Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1), 2005. ISSN 03600300. doi: 10.1145/1057977.1057980. URL `http://dl.acm.org/citation.cfm?id=1057977.1057980`. - **Cited** on pages 101 and 128.

Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, 2002. - **Cited** on pages 28, 34 and 38.

Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *FAST*, number September, pages 1–16, 2007. URL `http://www.usenix.org/event/fast07/tech/schroeder/schroeder.pdf`. - **Cited** on pages 2, 7 and 96.

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems - SIGMETRICS '09*, page 193, New York, New York, USA, June 2009. ACM Press. ISBN 9781605585116. doi: 10.1145/1555349.1555372. URL `http://portal.acm.org/citation.cfm?id=1555349.1555372`. - **Cited** on pages 2, 7 and 96.

António Sousa, José Pereira, Francisco Moura, and Rui Oliveira. Optimistic total order in wide area networks. In *IEEE Symposium on Reliable Distributed Systems*, 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.3676http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1180188`. - **Cited** on pages 101, 124, 127 and 128.

Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Networking Transactions*, 11(1):17–32, 2003. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/TNET.2002.808407. - **Cited** on page 16.

Chunqiang Tang, Rong N. Chang, and Christopher Ward. Gocast: Gossip-enhanced overlay multicast for fast and dependable group communication. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, pages 140–149, Washington, DC, USA, 2005. IEEE Computer Society. - **Cited** on page 94.

Twitter Engineering. Murder: Fast datacenter code deploys using BitTorrent. `http://t.co/uo5rEN4`, September, 2012. - **Cited** on page 61.

Robbert van Renesse, Ken Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21:164–206, 2003. - **Cited** on page 63.

Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In *Proceedings of the 14th IEEE International Conference on Network Protocols*, ICNP, pages 2–11. IEEE Computer Society, 2006. - **Cited** on page 92.

Hakon Verespej and Joseph Pasquale. A Characterization of Node Uptime Distributions in the PlanetLab Test Bed. *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 203–208, October 2011. doi: 10.1109/SRDS.2011.32. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6076778http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6076778`. - **Cited** on pages 2, 7 and 96.

Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1), 2009. ISSN 00010782. doi: 10.1145/1435417.1435432. URL `http://dl.acm.org/ft_gateway.cfm?id=1435432&type=html`. - **Cited** on pages 2, 99 and 128.

S. Voulgaris, D. Gavidia, and M. Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005a. - **Cited** on page 17.

Spyros Voulgaris and Maarten van Steen. Hybrid dissemination: Adding determinism to probabilistic multicasting in large-scale p2p systems. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, Middleware, pages 389–409, New York, Inc. New York, NY, USA, 2007. Springer-Verlag. - **Cited** on page 93.

Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:197–217, 2005b. - **Cited** on page 86.

Spyros Voulgaris, Márk Jelasity, and Maarten Van Steen. A Robust and Scalable Peer-to-Peer Gossiping Protocol. In *Agents and Peer-to-Peer Computing*, volume 2872 of *Lecture Notes in Computer Science*, pages 47–58. Springer-Verlag Berlin, Heidelberg, 2005c. - **Cited** on page 17.

Spyros Voulgaris, Etienne Rivière, Anne-Marie Kermarrec, and Maarten van Steen. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *International Workshop on Peer-to-Peer Systems*, 2006. - **Cited** on page 25.

Feng Wang, Yongqiang Xiong, and Jiangchuan 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. ISSN 1045-9219. doi: 10.1109/TPDS.2009.77. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4967573`. - **Cited** on page 22.

S. Whittaker, L. Terveen, W. Hill, and L. Cherny. The dynamics of mass interaction. In *Conference on Computer supported cooperative work*, 1998. - **Cited** on pages 35 and 53.

Wikimedia Foundation. Wikipedia database dumps. `http://dumps.wikimedia.org/`, 2013. - **Cited** on page 37.

N.C. Wormald. Models of random regular graphs. *Surveys in combinatorics*, 276:239–298, 1999. - **Cited** on page 57.

Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.1577`. - **Cited** on page 16.

Shelley Zhuang, Ben Zhao, Anthony Joseph, Randy Katz, and John Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV, pages 11–20, New York, NY, USA, 2001. ACM. - **Cited** on page 16.