Gossip-based Protocols for Large-scale Distributed Systems

DSc Dissertation

Márk Jelasity

Szeged, 2013

Contents

Preface **1**

1 Gossip Protocol Basics **3**
1.1 Gossip and Epidemics . . . 3
1.2 Information dissemination . . . 4
1.2.1 The Problem . . . 5
1.2.2 Algorithms and Theoretical Notions . . . 5
1.2.3 Applications . . . 12
1.3 Aggregation . . . 13
1.3.1 Algorithms and Theoretical Notions . . . 14
1.3.2 Applications . . . 17
1.4 What is Gossip after all? . . . 18
1.4.1 Overlay Networks . . . 18
1.4.2 Prototype-based Gossip Definition . . . 19
1.5 Conclusions . . . 19
1.6 Further Reading . . . 20

2 Peer Sampling Service **23**
2.1 Introduction . . . 23
2.2 Peer-Sampling Service . . . 25
2.2.1 API . . . 25
2.2.2 Generic Protocol Description . . . 25
2.2.3 Design Space . . . 27
2.2.4 Known Protocols as Instantiations of the Model . . . 29
2.2.5 Implementation . . . 29
2.3 Local Randomness . . . 30
2.3.1 Experimental Settings . . . 31
2.3.2 Test Results . . . 31
2.3.3 Conclusions . . . 33
2.4 Global Randomness . . . 33
2.4.1 Properties of Degree Distribution . . . 34
2.4.2 Clustering and Path Lengths . . . 40
2.5 Fault Tolerance . . . 42
2.5.1 Catastrophic Failure . . . 42
2.5.2 Churn . . . 43
2.5.3 Trace-driven Churn Simulations . . . 46
2.6 Wide-Area-Network Emulation . . . 48
2.7 Discussion . . . 50

2.7.1 Randomness . . . 50
2.8 Related Work . . . 52
2.8.1 Gossip Membership Protocols . . . 52
2.8.2 Complex Networks . . . 52
2.8.3 Unstructured Overlays . . . 53
2.8.4 Structured Overlays . . . 53
2.9 Concluding Remarks . . . 53

3 Average Calculation **55**
3.1 Introduction . . . 55
3.2 System Model . . . 56
3.3 Gossip-based Aggregation . . . 57
3.3.1 The Basic Aggregation Protocol . . . 57
3.3.2 Theoretical Analysis of Gossip-based Aggregation . . . 58
3.4 A Practical Protocol for Gossip-based Aggregation . . . 67
3.4.1 Automatic Restarting . . . 67
3.4.2 Coping with Churn . . . 67
3.4.3 Synchronization . . . 67
3.4.4 Importance of Overlay Network Topology for Aggregation . . . 68
3.4.5 Cost Analysis . . . 71
3.5 Aggregation Beyond Averaging . . . 72
3.5.1 Examples of Supported Aggregates . . . 72
3.5.2 Dynamic Queries . . . 75
3.6 Theoretical Results for Benign Failures . . . 75
3.6.1 Crashing Nodes . . . 75
3.6.2 Link Failures . . . 76
3.6.3 Conclusions . . . 78
3.7 Simulation Results for Benign Failures . . . 78
3.7.1 Node Crashes . . . 78
3.7.2 Link Failures and Message Omissions . . . 80
3.7.3 Robustness via Multiple Instances of Aggregation . . . 81
3.8 Experimental Results on PlanetLab . . . 82
3.9 Related Work . . . 85
3.10 Conclusions . . . 86

4 Distributed Power Iteration **89**
4.1 Introduction . . . 89
4.2 Chaotic Asynchronous Power Iteration . . . 90
4.3 Adding Normalization . . . 91
4.4 Controlling the Vector Norm . . . 92
4.4.1 Keeping the Vector Norm Constant . . . 92
4.4.2 The Random Surfer Operator of PageRank . . . 93
4.5 Experimental Results . . . 93
4.5.1 Notes on the Implementation . . . 93
4.5.2 Artificially Generated Matrices . . . 94
4.5.3 Results . . . 95
4.5.4 PageRank on WWW Crawl Data . . . 97
4.6 Related Work . . . 98

4.7 Conclusions . . . 98

5 Slicing Overlay Networks **99**
5.1 Introduction . . . 99
5.2 Problem Definition . . . 100
5.2.1 System Model . . . 100
5.2.2 The Ordered Slicing Problem . . . 100
5.3 A Gossip-based Approach . . . 101
5.4 Analogy with Gossip-based Averaging . . . 103
5.5 Experimental Analysis . . . 105
5.5.1 The Number of Successful Swaps . . . 105
5.5.2 Message Drop . . . 106
5.5.3 Churn . . . 107
5.5.4 An Illustrative Example . . . 108
5.6 Conclusions . . . 110

6 T-Man: Topology Construction **111**
6.1 Introduction . . . 111
6.2 Related Work and Contribution . . . 112
6.3 System Model . . . 113
6.4 The Overlay Construction Problem . . . 114
6.5 The T-MAN Protocol . . . 116
6.6 Key Properties of the Protocol . . . 117
6.6.1 Analogy with the Anti-Entropy Epidemic Protocol . . . 118
6.6.2 Parameter Setting for Symmetric Target Graphs . . . 118
6.6.3 Notes on Asymmetric Target Graphs . . . 120
6.6.4 Storage Complexity Analysis . . . 122
6.7 Experimental Results . . . 125
6.7.1 A Practical Implementation . . . 125
6.7.2 Simulation Environment . . . 127
6.7.3 Ranking Methods . . . 128
6.7.4 Performance Measures . . . 129
6.7.5 Evaluating the Starting Mechanism . . . 129
6.7.6 Evaluating the Termination Mechanism . . . 130
6.7.7 Parameter Tuning . . . 131
6.7.8 Failures . . . 132
6.8 Conclusions . . . 134

7 Bootstrapping Chord **135**
7.1 Introduction . . . 135
7.2 System Model . . . 136
7.3 The T-CHORD protocol . . . 136
7.3.1 A Brief Introduction to Chord . . . 137
7.3.2 T-CHORD . . . 137
7.3.3 T-CHORD-PROX: Network Proximity . . . 138
7.4 Experimental Results . . . 138
7.4.1 Experimental Settings . . . 138
7.4.2 Convergence . . . 139
7.4.3 Scalability . . . 140

7.4.4 Parameters 141
7.4.5 Robustness 141
7.4.6 Starting and Termination 142
7.5 Related Work 143
7.6 Conclusions 144

8 Towards a Generic Bootstrapping Service **145**
8.1 Introduction 145
8.2 The Architecture 146
8.3 Bootstrapping Prefix Tables 147
8.4 Simulation Results 148
8.5 Conclusions 150

Bibliography **151**

Preface

This dissertation is based on my work related to gossip protocols that solve various problems in massively distributed large scale networks. Resisting the temptation to present all my results in the area, my aim was to paint a coherent picture focusing on a subset of my core results consisting of closely related algorithms and systems that strongly build on each other. These results can be considered puzzle pieces that can be used to build a class of self-organizing massively distributed and robust adaptive systems.

Complying with the requirement of the Hungarian Academy of Sciences, the results included in the dissertation all originate from a period well after defending my PhD dissertation in the year 2001. Besides, my PhD dissertation was in the area of heuristic optimization and genetic algorithms, so there is no overlap with the presented work at all. In the following I present the outline of the dissertation, referring to the publications that form the basis of each chapter. I will describe my own contribution to these publications and I will mention results that are not covered in the dissertation when appropriate.

Chapter 1 is based mainly on [1], a textbook chapter I wrote to introduce gossip protocols. It was extended with material from [2], a survey paper that was written under my direction, based on my insight, namely that gossip protocols cannot be defined rigorously, instead they represent a design philosophy that one can follow to varying degrees and that is shared by many other approaches. The chapter was also revised, most importantly, the message passing formulations of the algorithms were improved so that they can form the basis of the common conventions that I adopted throughout the dissertation.

Chapter 2 is based on [3]. The origins of this work can be traced back to the NEWSCAST protocol that was published only as a technical report [4]. However, even as a technical report it achieved a rather high impact (138 citations as of now in Google Scholar) and was recently reprinted as a book chapter [5]. After realizing that many similar protocols have been proposed independently of NEWSCAST, the original idea underlying the unifying peer sampling framework was mine. This first resulted in a conference publication [6] followed by the journal article [3]. I am the main author of the paper and I contributed most of the implementation and experimental work as well (except the trace-driven simulations, and the cluster implementation and experiments). The presentation of the algorithm itself in this dissertation was thoroughly revised and was aligned with the common conventions that are based on an asynchronous event-based approach. I also added a section about the related protocols, in particular, NEWSCAST that inspired the unified framework. One notable publication not covered in the dissertation is [7] that presents an improvement over NEWSCAST to make it adaptive to different non-uniform localized patterns of failure and application load.

Chapter 3 is based on a journal article [8] that in turn is based on two conference publications: [9] and [10]. The idea of the algorithm, as well as the theoretical analysis is my contribution. I also implemented the algorithm, and contributed some of the experimental evaluation as well. In this chapter the theoretical analysis was *completely rewritten from*

scratch. This is due to the unfortunate fact that in the original publication the analysis was not correct. The new theoretical discussion leaves the main conclusions unchanged but provides rigorous formulations and proofs for the original results. This way I now precisely characterize the convergence speed in many interesting cases.

Chapter 4 is based on [11]. As the main author, I contributed the implementation and the experimental evaluation, as well as most details of the algorithm. This chapter illustrates how the peer sampling service from Chapter 2 and averaging from Chapter 3 can be combined to solve a practically relevant non-trivial problem. As in the other chapters, the algorithm presentation was thoroughly revised and aligned with the conventions.

Chapter 5 is based on [12]. The idea of the algorithm, its implementation, as well as its theoretical and experimental evaluation is my contribution. The interesting aspect of this work is that it implements a sorting algorithm in a distributed way that can be characterized using the theoretical tools developed in Chapter 3. Since there the theoretical results are new (as mentioned above), in this chapter the theoretical results are also thoroughly revised and extended, and a closer connection is made with averaging than what was presented in the original publication. The workshop paper [13] should be mentioned here as well (not covered in the dissertation) that implements an entirely different method for distributed ranking, that is also based on gossip.

Chapter 6 is based on the journal article [14], which in turn roots back to [15]. The original idea of the T-MAN algorithm and its first implementation is my contribution. My co-authors contributed practical features such as starting and termination variants. The approximative theoretical models and the related empirical analysis were also contributed by me. A part of the experimental work was completed by me as well. In this chapter the algorithm description was reworked to fit into the framework used by the other chapters, and the theoretical discussion was slightly reformulated and clarified.

The remaining two chapters discuss applications of T-MAN. Chapter 7 is based on [16] and partly also on [14]. Here I contributed to the design of the algorithm. The presentation of the algorithm was thoroughly revised to match the structure of the dissertation, and the experimental section was extended with results from [14]. Chapter 8 is based on [17], where the algorithm is based on my initial idea, and its implementation, and the experimental evaluation is my contribution. We note that this chapter also illustrates how to build complex applications from the components presented in the dissertation.

Finally, it should be noted here that this dissertation was completed while visiting Cornell University. This allowed me to fully focus on finalizing this work in an exceptional intellectual atmosphere, which clearly had a noticeable impact on the quality. I am especially grateful to Prof. Kenneth Birman—my host during the visit—who was very supportive of my efforts to finish the dissertation.

April 2013, Szeged, Hungary and Ithaca, NY, USA Márk Jelasity

Chapter 1

Gossip Protocol Basics

Gossip plays a very significant role in human society. Information spreads throughout the human grapevine at an amazing speed, often reaching almost everyone in a community, without any central coordinator. Moreover, rumor tends to be extremely stubborn: once spread, it is nearly impossible to erase it. In many distributed computer systems—most notably in *cloud computing* and *peer-to-peer computing*—this speed and robustness, combined with algorithmic simplicity and the lack of central management, are very attractive features.

Accordingly, over the past few decades several gossip-based algorithms have been developed to solve various problems. The prototypical application of gossip is information spreading (also known as multicast) where a piece of news is being spread over a large network. It the dissertation, this application is not discussed. However, since in this chapter our goal is to provide the necessary background, intuition and motivation for the gossip approach, we provide a brief introduction to this area.

After discussing information spreading, we move on to generalize the family of gossip protocols. To illustrate the generality of the gossip approach, we first discuss information aggregation (an area of distributed data mining), where distributed information is being summarized. We then present a completely generic gossip algorithm framework, that will accommodate most of the work in the dissertation.

1.1 Gossip and Epidemics

Like it or not, gossip plays a key role in human society. In his controversial book, Dunbar (an anthropologist) goes as far as to claim that the primary reason for the emergence of language was to permit gossip, which had to replace grooming—a common social reinforcement activity in primates—due to the increased group size of early human populations in which grooming was no longer feasible [18].

Whatever the case, it is beyond any doubt that gossip—apart from still being primarily a social activity—is highly effective in spreading information. In particular, information spreads very quickly, and the process is most resistant to attempts to stop it. In fact, sometimes it is so much so that it can cause serious damage; especially to big corporations. Rumors associating certain corporations to Satanism, or claiming that certain restaurant-chains sell burgers containing rat meat or milk shakes containing cow eyeball fluid as thickener, etc., are not uncommon. Accordingly, controlling gossip has long been an important area of research. The book by Kimmel gives many examples and details on human gossip [19].

While gossip is normally considered to be a means for spreading information, in reality information is not just transmitted mechanically but also processed. A person collects information, processes it, and passes the processed information on. In the simplest case, information is filtered at least for its degree of interest. This results in the most interesting pieces of news reaching the entire group, whereas the less interesting ones will stop spreading before getting to everyone. More complicated scenarios are not uncommon either, where information is gradually altered. This increases the complexity of the process and might result in emergent behavior where the community acts as a "collectively intelligent" (or sometimes perhaps not so intelligent) information processing medium.

Gossip is analogous to an epidemic, where a virus plays the role of a piece of information, and infection plays the role of learning about the information. In the past years we even had to learn concepts such as "viral marketing", made possible through Web 2.0 platforms such as video sharing sites, where advertisers consciously exploit the increasingly efficient and extended social networks to spread ads via gossip. The key idea is that shocking or very funny ads are especially designed so as to maximize the chances that viewers inform their friends about it, and so on.

Not surprisingly, epidemic spreading has similar properties to gossip, and is equally (if not more) important to understand and control. Due to this analogy and following common practice we will mix epidemiological and gossip terminology, and apply epidemic spreading theory to gossip systems.

Gossip and epidemics are of interest for large scale distributed systems for at least two reasons. The first reason is inspiration to design new protocols: gossip has several attractive properties like simplicity, speed, robustness, and a lack of central control and bottlenecks. These properties are very important for information dissemination and collective information processing (aggregation) that are both key components of large scale distributed systems.

The second reason is security research. With the steady growth of the Internet, viruses and worms have become increasingly sophisticated in their spreading strategies. Infected computers typically organize into networks (called botnets) and, being able to cooperate and perform coordinated attacks, they represent a very significant threat to IT infrastructure. One approach to fighting these networks is to try and prevent them from spreading, which requires a good understanding of epidemics over the Internet.

In this chapter we focus on the former aspect of gossip and epidemics: we treat them as inspiration for the design of robust self-organizing systems and services.

1.2 Information dissemination

The most natural application of gossip (or epidemics) in computer systems is spreading information. The basic idea of processes periodically communicating with peers and exchanging information is not uncommon in large scale distributed systems, and has been applied from the early days of the Internet. For example, the Usenet newsgroup servers spread posts using a similar method, and the IRC chat protocol applies a similar principle as well among IRC servers. In many routing protocols we can also observe routers communicating with neighboring routers and exchanging traffic information, thereby improving routing tables.

However, the first real application of gossip, that was based on theory and careful analysis, and that boosted scientific research into the family of gossip protocols, was part of a distributed database system of the Xerox Corporation, and was used to make sure each

replica of the database on the Xerox internal network was up-to-date [20]. In this section we will employ this application as a motivating example and illustration, and at the same time introduce several variants of gossip-based information dissemination algorithms.

1.2.1 The Problem

Let us assume we have a set of database servers (in the case of Xerox, 300 of them, but this number could be much larger as well). All of these servers accept updates; that is, new records or modifications of existing records. We want to inform all the servers about each update so that all the replicas of the database are identical and up-to-date.

Obviously, we need an algorithm to inform all the servers about a given update. We shall call this task *update spreading*. In addition, we should take into account the fact that whatever algorithm we use for spreading the update, it will not work perfectly, so we need a mechanism for *error correction*.

At Xerox, update spreading was originally solved by sending the update via email to all the servers, and error correction was done by hand. Sending emails is clearly not scalable: the sending node is a bottleneck. Moreover, multiple sources of error are possible: the sender can have an incomplete list of servers in the network, some of the servers can temporarily be unavailable, email queues can overflow, and so on.

Both tasks can be solved in a more scalable and reliable way using an appropriate (separate) gossip algorithm. In the following we first introduce several gossip models and algorithms, and then we explain how the various algorithms can be applied to solve the above mentioned problems.

1.2.2 Algorithms and Theoretical Notions

We assume that we are given a set of nodes that are able to pass messages to each other. In this section we will focus on the cost of spreading a single update among these nodes. That is, we assume that at a certain point in time, one of the nodes gets a new update from an external source, and from that point we are interested in the dynamics of the spreading of that update when using the algorithms we describe.

When discussing algorithms and theoretical models, we will use the terminology of epidemiology. According to this terminology, each node can be in one of three states, namely

- *susceptible (S)*: The node does not know about the update
- *infected (I)*: The node knows the update and is actively spreading it
- *removed (R)*: The node has seen the update, but is not participating in the spreading process (in epidemiology, this corresponds to death or immunity)

These states are relative to one fixed update. If there are several concurrent updates, one node can be infected with one update, while still being susceptible to another update, and so on.

In realistic applications there are typically many updates being propagated concurrently, and new updates are inserted continuously. Accordingly, our algorithms will in fact be formulated to deal with multiple updates that are coming continuously in an unpredictable manner. However, we present the simplest possible forms of these algorithms. It is important to note that additional techniques can be applied to optimize the amortized

Algorithm 1 SI gossip

```
 1: loop
 2:   wait(Δ)
 3:   p ← random peer
 4:   if push then
 5:     sendPush(p, known updates)
 6:   else if pull then
 7:     sendPullRequest(p)
 8:
 9: procedure ONPULLREQUEST(m)
10:   sendPull(m.sender, known updates)
11: procedure ONPUSH(m)
12:   if pull then
13:     sendPull(m.sender, known updates)
14:   store m.updates
15:
16: procedure ONPULL(m)
17:   store m.updates
```

cost of propagating a single update, when there are multiple concurrent updates in the system. In Section 1.2.3 we discuss some of these techniques. In addition, nodes might know the global list or even the insertion time of the updates, as well as the list of updates available at some other nodes. This information can also be applied to reduce propagation cost even further.

The allowed state transitions depend on the model that we study. Next, we shall consider the SI model and the SIR model. In the SI model, nodes are initially in state S with respect to a fixed update, and can change to state I (when they learn about the update). Once in state I, a node can no longer change its state (I is an absorbing state). In the SIR model, we allow nodes in state I to switch to state R, where R is the absorbing state. This means that in the SIR model nodes might stop spreading an update eventually, but they never forget about the update.

The Algorithm in the SI Model

The algorithm that implements gossip in the SI model is shown in Algorithm 1. It is formulated in an asynchronous message passing style, where each node executes one process (that we call the active thread) and, furthermore, it has message handlers that process incoming messages.

The active thread is executed once in each Δ time units. We will call this waiting period a *gossip cycle* (other terminology is also used such as gossip round or period).

In line 3 we assume that a node can select a random peer node from the set of all nodes. This assumption is not trivial, especially in very large and dynamically changing networks. In fact, peer sampling is a fundamental service that all gossip protocols rely on. We will discuss random peer sampling briefly in Section 1.4. Chapter 2 discusses random peer sampling in detail.

The algorithm makes use of two important Boolean parameters called *push* and *pull*. At least one of them has to be true, otherwise no messages are sent. Depending on these parameters, we can talk about push, pull, and push-pull gossip, each having significantly different dynamics and cost. In push gossip, susceptible nodes are passive and infective nodes actively infect the population. In pull and push-pull gossip each node is active.

Obviously, a node cannot stop pulling for updates unless it knows what updates can be expected; and it cannot avoid getting known updates either unless it advertises which updates it has already. As mentioned before, we present only a simple formulation: we pull continuously and we keep pushing all known updates as well. Practical applications will involve various techniques to minimize the redundant messages; although if the updates

Algorithm 2 SI gossip, simpler, but inferior version

```
1: loop
2:   wait(Δ)
3:   p ← random peer
4:   if push then
5:     sendUpdate(p, known updates)
6:   if pull then
7:     sendUpdateRequest(p)
8: procedure ONUPDATE(m)
9:   store m.updates
10:
11: procedure ONUPDATEREQUEST(m)
12:   sendUpdate(m.sender, known updates)
```

themselves are small, then in the SI model there is not much room for optimization.

We did in fact apply a form of optimization though. To see how, let us consider Algorithm 2. This algorithm is simpler and slightly more intuitive than Algorithm 1 but it is not identical: the difference is that in Algorithm 1 in the message handler ONPUSH we can explicitly control the order of processing the push message and sending the pull message when the push-pull variant is being run. In this case, it makes more sense to first send the pull message and then store the received updates, because this way some redundancy can be avoided. In fact we can easily make sure we send only the non-redundant updates back (we do not indicate this in the pseudocode to keep it simple).

Algorithm 2 does not offer this possibility of control (note that message delay is not under our control). For this reason, in the remaining parts of the thesis we will always use the style of formulation of Algorithm 1.

Basic Theoretical Properties of the SI Model

For theoretical purposes we will assume that messages are transmitted without delay, and for now we will assume that no failures occur in the system. We will also assume that messages are sent at the same time at each node, that is, messages from different cycles do not mix and cycles are synchronized. None of these assumptions are critical for practical usability, but they are needed for theoretical derivations that nevertheless give a fair indication of the qualitative and also quantitative behavior of gossip protocols.

Let us start with the discussion of the push model. We will consider the propagation speed of the update as a function of the number of nodes N. Let s_0 denote the proportion of susceptible nodes at the time of introducing the update at one node. Clearly, $s_0 = (N-1)/N$. Let s_t denote the proportion of susceptible nodes at the end of the t-th cycle; that is, at time $t\Delta$. We can calculate the expectation of s_{t+1} as a function of s_t, provided that the peer selected in line 3 is chosen independently at each node and independently of past decisions as well. In this case, we have

$$E(s_{t+1}) = s_t \left(1 - \frac{1}{N}\right)^{N(1-s_t)} \approx s_t e^{-(1-s_t)}, \tag{1.1}$$

where $N(1-s_t)$ is the number of nodes that are infected at cycle t, and $(1-1/N)$ is the probability that a fixed infected node will not infect some fixed susceptible node. Clearly, a node is susceptible in cycle $t+1$ if it was susceptible in cycle t and all the infected nodes picked some other node. Actually, as it turns out, this approximative model is rather accurate (the deviation from it is small), as shown by Pittel in [21]: we can take the expected value $E(s_{t+1})$ as a good approximation of s_{t+1}.

It is easy to see that if we wait long enough, then eventually all the nodes will receive the update. In other words, the probability that a particular node never receives the update

is zero. But what about the number of cycles that are necessary to let every node know about the update (become infected)? Pittel proves that in probability,

$$S_N = \log_2 N + \log N + O(1) \quad \text{as} \quad N \to \infty, \tag{1.2}$$

where $S_N = min\{t : s_t = 0\}$ is the number of cycles needed to spread the update.

The proof is rather long and technical, but the intuitive explanation is rather simple. In the initial cycles, most nodes are susceptible. In this phase, the number of infected nodes will double in each cycle to a good approximation. However, in the last cycles, where s_t is small, we can see from (1.1) that $E(s_{t+1}) \approx s_t e^{-1}$. This suggests that there is a first phase, lasting for approximately $\log_2 N$ cycles, and there is a last phase lasting for $\log N$ cycles. The "middle" phase, between these two phases, can be shown to be very fast, lasting a constant number of cycles.

Equation (1.2) is often cited as the key reason why gossip is considered efficient: it takes only $O(\log N)$ cycles to inform each node about an update, which suggests very good scalability. For example, with the original approach at Xerox, based on sending emails to every node, the time required is $O(N)$, assuming that the emails are sent sequentially.

However, let us consider the total number of messages that are being sent in the network until every node gets infected. For push gossip it can be shown that it is $O(N \log N)$. Intuitively, the last phase that lasts $O(\log N)$ cycles with s_t being very small already involves sending too many messages by the infected nodes. Most of these messages are in vain, since they target nodes that are already infected. The optimal number of messages is clearly $O(N)$, which is attained by the email approach.

Fortunately, the speed and message complexity of the push approach can be improved significantly using the pull technique. Let us consider s_t in the case of pull gossip. Here, we get the simple formula of

$$E(s_{t+1}) = s_t \cdot s_t = s_t^2, \tag{1.3}$$

which intuitively indicates a quadratic convergence if we assume the variance of s_t is small. When s_t is large, it decreases slowly. In this phase the push approach clearly performs better. However, when s_t is small, the pull approach results in a significantly faster convergence than push. In fact, the quadratic convergence phase, roughly after $s_t < 0.5$, lasts only for $O(\log \log N)$ cycles, as can be easily verified.

One can, of course, combine push and pull. This can be expected to work faster than either push or pull separately, since in the initial phase push messages will guarantee fast spreading, while in the end phase pull messages will guarantee the infecting of the remaining nodes in a short time. Although faster in practice, the speed of push-pull is still $O(\log N)$, due to the initial exponential phase.

What about message complexity? Since in each cycle each node will send at least one request, and $O(\log N)$ cycles are necessary for the update to reach all the nodes, the message complexity is $O(N \log N)$. However, if we count only the updates, and ignore request messages, we get a different picture. Just counting the updates is not meaningless, because an update message is normally orders of magnitude larger than a request message. It has been shown that in fact the push-pull gossip protocol sends only $O(N \log \log N)$ updates in total [22].

The basic idea behind the proof is again based on dividing the spreading process into phases and calculating the message complexity and duration of each phase. In essence,

the initial exponential phase—that we have seen with push as well—requires only $O(N)$ update transmissions, since the number of infected nodes (that send the messages) grows exponentially. But the last phase, the quadratic shrinking phase as seen with pull, lasts only $O(\log \log N)$ cycles. Needless to say, as with the other theoretical results, the mathematical proof is quite long and technical.

The SIR Model

In the previous section we outlined some important theoretical results regarding convergence speed and message complexity. However, we ignored one problem that can turn out to be important in practical scenarios: termination.

Push protocols never terminate in the SI model, constantly sending useless updates even after each node has received every update. Pull protocols could stop sending messages *if* the complete list of updates was known in advance: after receiving all the updates, no more requests need to be sent. However, in practice not even pull protocols can terminate in the SI model, because the list of updates is rarely known.

Here we will discuss solutions to the termination problem in the SIR model. These solutions are invariably based on some form of detecting and acting upon the "age" of the update.

We can design our algorithm with two different goals in mind. First, we might wish to ensure that the termination is optimal; that is, we want to inform all the nodes about the update, and we might want to minimize redundant update transmissions at the same time. Second, we might wish to opt for a less intelligent, simple protocol and analyze the size of the proportion of the nodes that will not get the update as a function of certain parameters.

One simple way of achieving the first design goal of optimality is by keeping track of the age of the update explicitly, and stop transmission (i.e., switching to the removed state, hence implementing the SIR model) when a pre-specified age is reached. This age threshold must be calculated to be optimal for a given network size N using the theoretical results sketched above. This, of course, assumes that each node knows N. In addition, a practically error- and delay-free transmission is also assumed, or at least a good model of the actual transmission errors is needed.

Apart from this problem, keeping track of the age of the update explicitly represents another, non-trivial practical problem. We assumed in our theoretical discussions that messages have no delay and that cycles are synchronized. When these assumptions are violated, it becomes rather difficult to determine the age of an update with an acceptable precision.

From this point on, we shall discard this approach, and focus on simple asynchronous methods that are much more robust and general, but are not optimal. To achieve the second design goal of simplicity combined with reasonable performance, we can try to guess when to stop based on local information and perhaps information collected from a handful of peers. These algorithms have the advantage of simplicity and locality. Besides, in many applications of the SIR model, strong guarantees on complete dissemination are not necessary, as we will see later on.

Perhaps the simplest possible implementation is when a node moves to the removed state with a fixed probability whenever it encounters a peer that has already received the update. Let this probability be $1/k$, where the natural interpretation of parameter k is the average number of times a node sends the update to a peer that turns out to already have

Algorithm 3 an SIR gossip variant

```
 1: loop
 2:    wait(Δ)
 3:    p ← random peer
 4:    if push then
 5:       sendPush(p, infective updates)
 6:    else if pull then
 7:       sendPullRequest(p)
 8:
 9: procedure ONFEEDBACK(m)
10:    for all u ∈ m.updates do
11:       switch u to state R with pr. 1/k
12: procedure ONPUSH(m)
13:    if pull then
14:       sendPull(m.sender, infective updates)
15:    onPull(m)
16:
17: procedure ONPULL(m)
18:    buffer ← m.updates ∩ {known updates}
19:    sendFeedback(m.sender, buffer)
20:    store m.updates
21:
22: procedure ONPULLREQUEST(m)
23:    sendPull(m.sender, infective updates)
```

the update before stopping its transmission. Obviously, this implicitly assumes a feedback mechanism because nodes need to check whether the peer they sent the update to already knew the update or not.

As shown in Algorithm 3, this feedback mechanism is the only difference between SIR and SI gossip, apart from the fact that in the SI model all known updates are infective, whereas in the SIR model they are either infective or removed. The active thread and procedure ONPULLREQUEST are identical to Algorithm 1. However, procedures ONPUSH and ONPULL send a feedback containing the received updates that were known already. This message is processed by procedure ONFEEDBACK, eventually switching all updates to the removed state. Removed updates are stored but are not included in the push and pull messages anymore.

A typical approach to model the SIR algorithm is to work with differential equations, as opposed to the discrete stochastic approach we applied previously. Let us illustrate this approach via an analysis of Algorithm 3, assuming a push variant. Following [20, 23], we can write

$$\frac{ds}{dt} = -si \tag{1.4}$$

$$\frac{di}{dt} = si - \frac{1}{k}(1-s)i \tag{1.5}$$

where $s(t)$ and $i(t)$ are the proportions of susceptible and infected nodes, respectively. The nodes in the removed state are given by $r(t) = 1 - s(t) - i(t)$. We can take the ratio, eliminating t:

$$\frac{di}{ds} = -\frac{k+1}{k} + \frac{1}{ks}, \tag{1.6}$$

which yields

$$i(s) = -\frac{k+1}{k}s + \frac{1}{k}\log s + c, \tag{1.7}$$

where c is the constant of integration, which can be determined using the initial condition that $i(1 - 1/N) = 1/N$ (where N is the number of nodes). For a large N, we have $c \approx (k+1)/k$.

Now we are interested in the value s^* where $i(s^*) = 0$: at that time sending the update is terminated, because all nodes are susceptible or removed. In other words, s^* is the

proportion of nodes that do not know the update when gossip stops. Ideally, s^* should be zero. Using the results, we can write an implicit equation for s^* as follows:

$$s^* = \exp[-(k+1)(1-s^*)]. \tag{1.8}$$

This tells us that the spreading is very effective. For $k = 1$, 20% of the nodes are predicted to miss the update, but with $k = 5$, 0.24% will miss it, while with $k = 10$ it will be as few as 0.00017%.

Let us now proceed to discussing message complexity. Since full dissemination is not achieved in general, our goal is now to approximate the number of messages needed to decrease the proportion of susceptible nodes to a specified level.

Let us first consider the push variant. In this case, we make the rather striking observation that the value of s depends only on the number of messages m that have been sent by the nodes. Indeed, each infected node picks peers independently at random to send the update to. That is, every single update message is sent to a node selected independently at random from the set of all the nodes. This means that the probability that a fixed node is in state S after a total of m update messages has been sent can be approximated by

$$s(m) = (1 - \frac{1}{N})^m \approx \exp[-\frac{m}{N}] \tag{1.9}$$

Substituting the desired value of s, we can easily calculate the total number of messages that need to be sent in the system: it is

$$m \approx -N \log s \tag{1.10}$$

If we demand that $s = 1/N$, that is, we allow only for a single node not to see the update, then we need $m \approx N \log N$. This reminds us of the SI model, that had an $O(N \log N)$ message complexity to achieve full dissemination. If, on the other hand, we allow for a constant proportion of the nodes not to see the update ($s = 1/c$) then we have $m \approx N \log c$; that is, a linear number of messages suffice. Note that s or m cannot be set directly, but only through other parameters such as k.

Another notable point is that (1.9) holds irrespective of whether we apply a feedback mechanism or not, and irrespective of the exact algorithm applied to switch to state R. In fact, it applies even for the pure SI model, since all we assumed was that it is a push-only gossip with random peer selection. Hence it is a strikingly simple, alternative way to illustrate the $O(N \log N)$ message complexity result shown for the SI model: roughly speaking, we need approximately $N \log N$ messages to make s go below $1/N$.

Since m determines s irrespective of the details of the applied push gossip algorithm, the speed at which an algorithm can have the infected nodes send m messages determines the speed of convergence of s. With this observation in mind, let us compare a number of variants of SIR gossip.

Apart from Algorithm 3, one can implement termination (switching to state S) in several different ways. For example, instead of a probabilistic decision in procedure ON-FEEDBACK, it is also possible to use a counter, and switch to state S after receiving the k-th feedback message. Feedback could be eliminated altogether, and moving to state R could depend only on the number of times a node has sent the update.

It is not hard to see that the the counter variants improve load balancing. This in turn improves speed because we can always send more messages in a fixed amount of time if the message sending load is well balanced. In fact, among the variants described

above, applying a counter without feedback results in the fastest convergence. However, parameter k has to be set appropriately to achieve a desired level of s. To set k and s appropriately, one needs to know the network size. Variants using a feedback mechanism achieve a somewhat less efficient load balancing but they are more robust to the value of k and to network size: they can "self-tune" the number of messages based on the feedback. For example, if the network is large, more update messages will be successful before the first feedback is received.

Lastly, as in the SI model, it is apparent that in the end phase the pull variant is much faster and uses fewer update messages. It does this at the cost of constantly sending update requests.

We think in general that, especially when updates are constantly being injected, the push-pull algorithm with counter and feedback is probably the most desirable alternative.

1.2.3 Applications

We first explain how the various protocols we discussed were applied at Xerox for maintaining a consistent set of replicas of a database. Although we cannot provide a complete picture here (see [20]), we elucidate the most important ideas.

In Section 1.2.1 we identified two sub-problems, namely update spreading and error correction. The former is implemented by an SIR gossip protocol, and the latter by an SI protocol. The SIR gossip is called *rumor mongering* and is run when a new update enters the system. Note that in practice, many fresh updates can piggyback a single gossip message, but the above-mentioned convergence properties hold for any single fixed update.

The SI algorithm for error correction works for every update ever entered, irrespective of age, simultaneously for all updates. In a naive implementation, the entire database would be transmitted in each cycle by each node. Evidently, this is not a good idea, since databases can be very large, and are mostly rather similar. Instead, the nodes first try to discover what the difference is between their local replicas by exchanging compressed descriptions such as checksums (or lists of checksums taken at different times) and transmit only the missing updates. However, one cycle of error correction is typically much more expensive than rumor mongering.

The SI algorithm for error correction is called *anti-entropy*. This is not a very fortunate name: we should remark here that it has no deeper meaning than to express the fact that "anti-entropy" will increase the similarity among the replicas thereby increasing "order" (decreasing randomness). So, since entropy is usually considered to be a measure of "disorder", the name "anti-entropy" simply means "anti-disorder" in this context.

In the complete system, the new updates are spread through rumor mongering, and anti-entropy is run occasionally to take care of any undelivered updates. When such an undelivered update is found, the given update is redistributed by re-inserting it as a new update into the database where it was not present. This is a very simple and efficient method, because update spreading via rumor mongering has a cost that depends on the number of other nodes that already have the update: if most of the nodes already have it, then the redistribution will die out very quickly.

Let us quickly compare this solution to the earlier, email based approach. Emailing updates and rumor mongering are similar in that both focus on spreading a single update and have a certain small probability of error. Unlike email, gossip has no bottleneck nodes and hence is less sensitive to local failure and assumes less about local resources such as

bandwidth. This makes gossip a significantly more scalable solution. Gossip uses slightly more messages in total for the distribution of a single update. But with frequent updates in a large set of replicas, the amortized cost of gossip (number of messages per update) is more favorable (remember that one message may contain many updates).

In practical implementations, additional significant optimizations were performed. Perhaps the most interesting one is *spatial gossip* where, instead of picking a peer at random, nodes select peers based on a distance metric. This is important because if the underlying physical network topology is such that there are bottleneck links connecting dense clusters, then random communication places a heavy load on such links that grows linearly with system size. In spatial gossip, nodes favor peers that are closer in the topology, thereby relieving the load from long distance links, but at the same time sacrificing some of the spreading speed. This topic is discussed at great length in [24].

We should also mention the removal of database entries. This is solved through "death certificates" that are updates stating that a given entry should be removed. Needless to say, death certificates cannot be stored indefinitely because eventually the databases would be overloaded by them. This problem requires additional tricks such as removing most but not all of them, so that the death certificate can be reactivated if the removed update pops up again.

Apart from the application discussed above, the gossip paradigm has recently received yet another boost. After getting used to Grid and P2P applications, and witnessing the emergence of the huge, and often geographically distributed data centers that increase in size and capacity at an incredible rate, in the past years we had to learn another term: *cloud computing* [25–27].

Cloud computing involves a huge amount of distributed resources (a cloud), typically owned by a single organization, and organized in such a way that for the user it appears to be a coherent and reliable storage or computing service. The exact details of commercially deployed technology are not always clear, but from several sources it seems rather evident that gossip protocols are involved. For example, after a recent crash of Amazon's S3 storage service, the message explaining the failure included some details:

> (...) Amazon S3 uses a gossip protocol to quickly spread server state information throughout the system. This allows Amazon S3 to quickly route around failed or unreachable servers, among other things.[1] (...)

In addition, a recent academic publication on the technology underlying Amazon's computing architecture provides further details on gossip protocols [28], revealing that an anti-entropy gossip protocol is responsible for maintaining a full membership table at each server (that is, a fully connected overlay network with server state information).

1.3 Aggregation

The gossip communication paradigm can be generalized to applications other than information dissemination. In these applications some implicit notion of spreading information will still be present, but the emphasis is not only on spreading but also on *processing* information on the fly.

This processing can be for creating summaries of distributed data; that is, computing a global function over the set of nodes based only on gossip-style communication. For

[1] `http://status.aws.amazon.com/s3-20080720.html`

Algorithm 4 push-pull averaging

```
1: loop
2:    wait(Δ)
3:    p ← random peer
4:    sendPush(p,x)
5: procedure ONPUSH(m)
6:    sendPull(m.sender,x)
7:    x ← (m.x + x)/2
8:
9: procedure ONPULL(m)
10:   x ← (m.x + x)/2
```

example, we might be interested in the average, or maximum of some attribute of the nodes. The problem of calculating such global functions is called data aggregation or simply *aggregation*. We might want to compute more complex functions as well, such as fitting models on fully distributed data, in which case we talk about the problem of *distributed data mining*.

In the past few years, a lot of effort has been directed at a specific problem: calculating averages. Averaging can be considered the archetypical example of aggregation. Chapter 3 will discuss this problem in detail, here we describe the basic notions to help illustrate the generality of the gossip approach.

Averaging is a very simple problem, and yet very useful: based on the average of a suitably defined local attribute, we can calculate a wide range of values. To elaborate on this notion, let us introduce some formalism. Let x_i be an attribute value at node i for all $0 < i \leq N$. We are interested in the average $\sum_{i=1}^{N} x_i/N$. Clearly, if we can calculate the average then we can calculate any mean of the form

$$g(x_1, \ldots, x_N) = f^{-1}\left(\frac{\sum_{i=1}^{N} f(x_i)}{N}\right) \tag{1.11}$$

as well, where we simply apply $f()$ on the local attributes before averaging. For example, $f(x) = \log x$ generates the geometric mean, while $f(x) = 1/x$ generates the harmonic mean. In addition, if we calculate the mean of several powers of x_i, then we can calculate the moments of the distribution of the values. For example, the variance can be expressed as a function over averages of x_i^2 and x_i:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - (\frac{1}{N}\sum_{i=1}^{N} x_i)^2 \tag{1.12}$$

Finally, other interesting quantities can be calculated using averaging as a primitive. For example, if every attribute value is zero, except at one node, where the value is 1, then the average is $1/N$. This allows us to compute the network size N.

In the remaining parts of this section we focus on several gossip protocols for calculating the average of node attributes.

1.3.1 Algorithms and Theoretical Notions

The first, perhaps simplest, algorithm we discuss is push-pull averaging, presented in Algorithm 4. Each node periodically selects a random peer to communicate with, and then sends the local estimate of the average x. The recipient node then replies with its own current estimate. Both participating nodes (the sender and the one that sends the reply) will store the average of the two previous estimates as a new estimate.

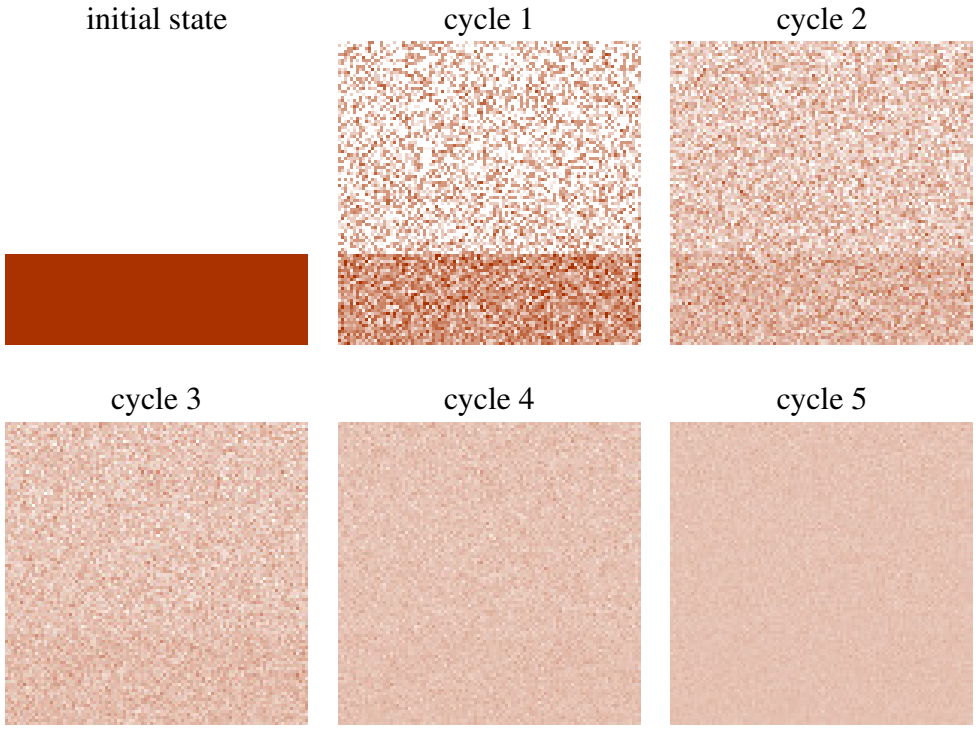


Figure 1.1: Illustration of the averaging protocol. Pixels correspond to nodes (100x100 pixels=10,000 nodes) and pixel color to the local approximation of the average.

Similarly to our treatment of information spreading, Algorithm 4 is formulated for an asynchronous message passing model, but we will assume several synchronicity properties when discussing the theoretical behavior of the algorithm. We will return to the issue of asynchrony in Section 1.3.1.

For now, we also treat the algorithm as a one-shot algorithm; that is, we assume that first the local estimate x_i of node i is initialized as $x_i = x_i(0)$ for all the nodes $i = 1 \dots N$, and subsequently the gossip algorithm is executed. This assumption will also be relaxed later in this section, where we briefly discuss the case, where the local attributes $x_i(0)$ can change over time and the task is to continuously update the approximation of the average.

Let us first have a brief look at the convergence of the algorithm. It is clear that the state when all the x_i values are identical is a fixed point, assuming there are no node failures and message failures, and that the messages are delivered without delay. In addition, observe that the *sum* of the approximations remains constant throughout. This very important property is called *mass conservation*. We can then look at the difference between the minimal and maximal approximations and show that this difference can only decrease and, furthermore, it converges to zero in probability, using the fact that peers are selected at random. But if all the approximations are the same, they can only be equal to the average $\sum_{i=1}^{N} x_i(0)/N$ due to mass conservation.

The really interesting question, however, is the *speed* of convergence. The fact of convergence is easy to prove in a probabilistic sense, but such a proof is useless from a practical point of view without characterizing speed. The speed of the protocol is illustrated in Figure 1.1. The process shows a diffusion-like behavior. The averaging algorithm is of course executed using random peer sampling (the pixel pairs are picked at random). The arrangement of the pixels is for illustration purposes only.

Algorithm 5 push averaging

```
1: loop
2:     wait(Δ)
3:     p ← random peer
4:     sendPush(p,(x/2, w/2))
5:     x ← x/2
6:     w ← w/2
7: procedure ONPUSH(m)
8:     x ← m.x + x
9:     w ← m.w + w
```

In Chapter 3 we characterize the speed of convergence and show that the variance of the approximations decreases by a constant factor in each cycle. In practice, 10-20 cycles of the protocol already provide an extremely accurate estimation: the protocol not only converges, but it converges very quickly as well.

Asynchrony

In the case of information dissemination, allowing for unpredictable and unbounded message delays (a key component of the asynchronous model) has no effect on the correctness of the protocol, it only has an (in practice, marginal) effect on spreading speed. For Algorithm 4 however, correctness is no longer guaranteed in the presence of message delays.

To see why, imagine that node j receives a PUSHUPDATE message from node i and as a result it modifies its own estimate and sends its own previous estimate back to i. But after that point, the mass conservation property of the network will be violated: the sum of all approximations will no longer be correct. This is not a problem if neither node j nor node i receives or sends another message during the time node i is waiting for the reply. However, if they do, then the state of the network may become corrupted. In other words, if the pair of push and pull messages are not *atomic*, asynchrony is not tolerated well.

Algorithm 5 is a clever modification of Algorithm 4 and is much more robust to message delay. The algorithm is very similar, but here we introduce another attribute called w. For each node i, we initially set $w_i = 1$ (so the sum of these values is N). We also modify the interpretation of the current estimate: on node i it will be x_i/w_i instead of x_i, as in the push-pull variant.

To understand why this algorithm is more robust to message delay, consider that we now have mass conservation in a different sense: the sum of the attribute values at the nodes *plus* the sum of the attribute values in the undelivered messages remains constant, for both attributes x and w. This is easy to see if one considers the active thread which keeps half of the values locally and sends the other half in a message. In addition, it can still be proven that the variance of the approximations x_i/w_i can only decrease.

As a consequence, messages can now be delayed, but if message delay is bounded, then the variance of the set of approximations at the nodes and in the messages waiting for delivery will tend to zero. Due to mass conservation, these approximations will converge to the true average, irrespective of how much of the total "mass" is in undelivered messages. (Note that the variance of x_i or w_i alone is not guaranteed to converge zero.)

Robustness to failure and dynamism

We will now consider message and node failures. Both kinds of failures are unfortunately more problematic than asynchrony. In the case of information dissemination, failure had

no effect on correctness: message failure only slows down the spreading process, and node failure is problematic only if every node fails that stores the new update.

In the case of push averaging, losing a message typically corrupts mass conservation. In the case of push-pull averaging, losing a push message will have no effect, but losing the reply (pull message) may corrupt mass conservation. The solutions to this problem are either based on failure detection (that is, they assume a node is able to detect whether a message was delivered or not) and correcting actions based on the detected failure, or they are based on a form of rejuvenation (restarting), where the protocol periodically re-initializes the estimates, thereby restoring the total mass. The restarting solution is feasible due to the quick convergence of the protocol. Both solutions are somewhat inelegant; but gossip is attractive mostly because of the lack of reliance on failure detection, which makes restarting more compatible with the overall gossip design philosophy. Unfortunately restarting still allows for a bounded inaccuracy due to message failures, while failure detection offers accurate mass conservation.

Node failures are a source of problems as well. By node failure we mean the situation when a node leaves the network without informing the other nodes about it. Since the current approximation x_i (or x_i/w_i) of a failed node i is typically different from $x_i(0)$, the set of remaining nodes will end up with an incorrect approximation of the average of the remaining attribute values. Handling node failures is problematic even if we assume perfect failure detectors. Solutions typically involve nodes storing the contributions of each node separately. For example, in the push-pull averaging protocol, node i would store δ_{ji}: the sum of the incremental contributions of node j to x_i. More precisely, when receiving an update from j (push or pull), node i calculates $\delta_{ji} = \delta_{ji} + (x_j - x_i)/2$. When node i detects that node j failed, it performs the correction $x_i = x_i - \delta_{ji}$.

We should mention that this is feasible only if the selected peers are from a small fixed set of neighboring nodes (and not randomly picked from the network), otherwise all the nodes would need to monitor an excessive number of other nodes for failure. Besides, message failure can interfere with this process too. The situation is further complicated by nodes failing temporarily, perhaps not even being aware of the fact that they have been unreachable for a long time by some nodes. Also note that the restart approach solves the node failure issue as well, without any extra effort or failure detectors, although, as previously, allowing for some inaccuracy.

Finally, let us consider a dynamic scenario where mass conservation is violated due to changing $x_i(0)$ values (so the approximations evolved at the nodes will no longer reflect the correct average). In such cases one can simply set $x_i = x_i + x_i^{new}(0) - x_i^{old}(0)$, which corrects the sum of the approximations, although the protocol will need some time to converge again. As in the previous cases, restarting solves this problem too without any extra measures.

1.3.2 Applications

The diffusion-based averaging protocols we focused on will most often be applied as a primitive to help other protocols and applications such as load balancing, task allocation, or the calculation of relatively complex models of distributed data such as spectral properties of the underlying graph [11, 29]. An example of this application will be described in Chapter 4.

Sensor networks are especially interesting targets for applications, due to the fact that their very purpose is data aggregation, and they are inherently local: nodes can typically

Algorithm 6 The gossip algorithm skeleton.

```
 1: loop
 2:    wait(Δ)
 3:    p ← selectPeer()
 4:    if push then
 5:       sendPush(p,state)
 6:    else if pull then
 7:       sendPullRequest(p)
 8:
 9: procedure ONPULLREQUEST(m)
10:    sendPull(m.sender,state)
11: procedure ONPUSH(m)
12:    if pull then
13:       sendPull(m.sender,state)
14:    state ← update(state,m.state)
15:
16: procedure ONPULL(m)
17:    state ← update(state,m.state)
```

communicate with their neighbors only [30]. However, sensor networks do not support point-to-point communication between arbitrary pairs of nodes as we assumed previously, which makes the speed of averaging slower, depending on the communication range of the devices.

1.4 What is Gossip after all?

So far we have discussed two applications of the gossip idea: information dissemination and aggregation. By now it should be rather evident that these applications, although different in detail, have a common algorithmic structure. In both cases an active thread selects a peer node to communicate with, followed by a message exchange and the update of the internal states of both nodes (for push-pull) or one node (for push or pull). We propose the *template* (or *design pattern* [31]) shown in Algorithm 6 to capture this structure. The three components that need to be defined to instantiate this pattern are methods UPDATE and SELECTPEER, and the state of a node. This template covers our two examples presented earlier. In the case of information dissemination the state of a node is defined by the stored updates, while in the case of averaging the state is the current approximation of the average at the node. In addition, the template covers a large number of other protocols as well.

1.4.1 Overlay Networks

To illustrate the power of this abstraction, we briefly mention one notable application we have not covered in this chapter: the construction and management of *overlay networks*. The larger part of this dissertation, in particular, Chapters 2 and 6 will discuss such applications. In this case the state of a node is a set of node addresses that define an overlay network. (A node is able to send messages to an address relying on lower layers of the networking stack; hence the name "overlay".)

In a nutshell, the state (the set of overlay links) is then communicated via gossip, and method UPDATE selects the new set of links from the set of all links the node has seen. Through this mechanism one can create and manage a number of different overlay networks such as random networks, structured networks (like a ring) or proximity networks based on some distance metric, for example semantic or latency-based distance. Method SELECTPEER can also be implemented in a clever way, based on the actual neighbors, to speed up convergence.

These networks can be applied by higher level applications, or by other gossip protocols. For example, random networks are excellent for implementing random peer sampling, a service all the algorithms rely on in this chapter when selecting a random peer to communicate with.

1.4.2 Prototype-based Gossip Definition

The gossip abstraction is powerful, perhaps too much so. It is rather hard to capture what this "gossipness" concept means exactly. Attempts have been made to define gossip formally, with mixed success [32]. For example, periodic and local communication to random peers appears to be a core feature. However, in the SIR model, nodes can stop communicating. Besides, in some gossip protocols neighboring nodes need not be random in every cycle but instead they can be fixed and static. For example, many secure gossip protocols in fact use deterministic peer selection on controlled networks [33]. Also, quite clearly, a protocol remains gossip if message sending is slightly irregular—for example, due to an optimization that makes a protocol adaptive to system load or the progress of information spreading. In general, the template allows us to model practically any message passing protocol, since the definition of state is unrestricted, in any cycle a peer can choose to send a zero length message (that is, no message), and the gossip period Δ can be arbitrarily small.

For this reason it appears to be more productive to also have a feature list that defines an idealized prototypical gossip protocol application (i.e., information spreading), and to compare the features of a given protocol with this set. In this way, instead of giving a formal, exact definition of gossip protocols, we make it possible to compare any given protocol to the prototypical gossip protocol and assess the similarities and differences, avoiding a rigid binary (gossip/non-gossip) decision over protocols. We propose the following features: (1) randomized peer selection, (2) only local information is available at all nodes, (3) cycle-based (periodic), (4) limited transmission and processing capacity per cycle, (5) all peers run the same algorithm.

The inherent and intentional fuzziness in this prototype-based approach turns the yes/no distinction of a formal definition into a measure of distance from prototypical gossip: a certain algorithm might have some of the properties, and might not have some others. Even in the case of matching properties, we can talk about the degree of matching. For example, we can ask how random peer selection is, or how local the decisions are.

Figure 1.2 is a simple illustration of this idea. The figure also illustrates the possibility that some algorithms from other fields might actually be closer to prototypical gossip than some protocols currently called gossip. The examples mentioned in the diagram are explained in detail in [2].

1.5 Conclusions

In this chapter we introduced the gossip design pattern through the examples of information dissemination (the prototypical application) and aggregation. We showed that both applications use a very similar communication model, and both applications provide probabilistic guarantees for an efficient and effective execution. We also discussed the gossip model in general, and briefly mentioned overlay network management as a further application.

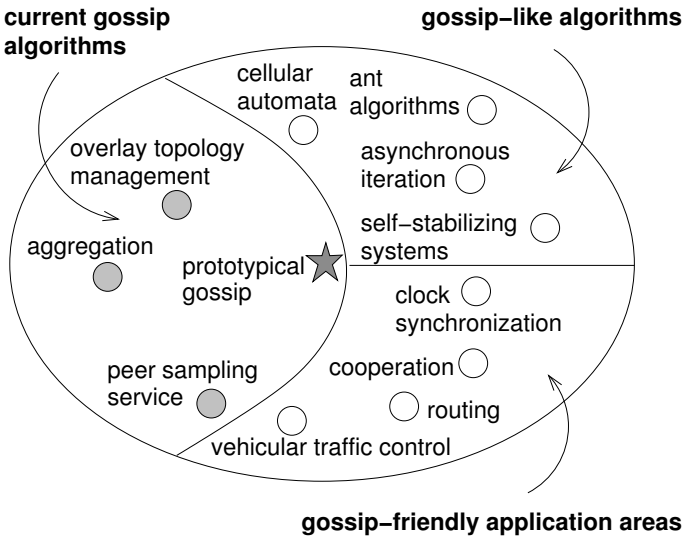


Figure 1.2: Prototypical gossip in a multidisciplinary context.

We should emphasize that gossip protocols represent a departure from "classical" approaches in distributed computing where correctness and reliability were the top priorities and performance (especially speed and responsiveness) was secondary. To put it simply: gossip protocols—if done well—are simple, fast, cheap, and extremely scalable, but they do not always provide a perfect or correct result under *all* circumstances and in *all* system models. But in many scenarios a "good enough" result is acceptable, and in realistic systems gossip components can always be backed up by more heavy-weight, but more reliable methods that provide eventual consistency or correctness.

A related problem is malicious behavior. Unfortunately, gossip protocols in open systems with multiple administrative domains are rather vulnerable to malicious behavior. Current applications of gossip are centered on single administrative domain systems, where all the nodes are controlled by the same entity and therefore the only sources of problems are hardware, software or network failures. In an open system nodes can be selfish, or even worse, they can be malicious. Current secure gossip algorithms are orders of magnitude more complicated than basic versions, thus losing many of the original advantages of gossiping.

All in all, gossip algorithms are a great tool for solving certain kinds of very important problems under certain assumptions. In particular, they can help in the building of enormous cloud computing systems that are considered the computing platform of the future by many, and provide tools for programming sensor networks as well.

1.6 Further Reading

Here we provide a number of references that might help the reader probe deeper into the topics that we discussed in this chapter. Our discussion on information dissemination was based mostly on the seminal paper of Demers et al. [20], and partly on [21] and [22] to elaborate on some of the details. In general [20] is highly recommended for further study, since the paper contains a lot more material than what was covered here, and it touches on almost all research issues associated with this field. One further important aspect—gossiping taking physical distance into account—is elaborated on in a paper by Kempe,

Kleinberg and Demers [24].

In the case of aggregation, we based our discussion on [8] (as discussed in Chapter 3) and [34], borrowing some ideas from [35] and [36] during the discussion of asynchrony, presented in a simplified form. Certain aspects of gossip aggregation have been covered in recent work such as increased fault tolerance [37, 38] or privacy preservation [39].

An alternative theoretical approach can be followed as well if gossip-based average calculation is viewed mathematically as a random walk on a suitably defined graph (or, equivalently, a Markovian process with a time-reversible Markovian transition matrix), which has a uniform stationary distribution. This however assumes that computation is performed synchronously, in lock step, so that averaging can be described as a matrix iteration converging to the uniform distribution. These approaches have been omitted here, but the interested reader can find them in a number of seminal papers such as [40–42]. In general, [43] gives an excellent and comprehensive tutorial on the relevant mathematical theory.

There are yet other alternative ways of calculating averages that have not been covered but that are based on gossip in one way or another. One approach is based on maintaining a hierarchical overlay (often involving gossip in the construction and maintenance of the hierarchy) and using it to calculate various aggregates [44–46]. This approach is rather typical in wireless sensor networks [47]. One can also use additional tricks such as random walk-based statistical approximations [48] or cleverly constructed attributes such that the averaging problem is reduced to finding the minima of these attributes, and then using these minima to infer the true average with a controlled accuracy [49]. A method inspired by belief propagation was also proposed, that has favorable properties in certain communication topologies [50].

Let us now suggest some further reading on applications of gossip that have not been covered in this chapter. In Section 1.4 we mentioned *peer sampling*, mentioning gossip-based implementations. An extensive discussion of peer sampling and its variations is provided in [3] and discussed in Chapter 2. A similar application (discussed in Chapter 6), is overlay construction and maintenance. We show that a wide range of network topologies can be evolved with a slight modification of the peer sampling algorithm.

Gossip has also been applied in fault tolerant, practical distributed hash table implementations [51] to increase the robustness of the overlay maintenance algorithm for dynamic conditions.

Chapter 2

Peer Sampling Service

As we have seen, Algorithm 6 in Chapter 1 crucially relies on a method called SELECTPEER. The *peer sampling service* implements this function. In short, this service provides every node with peers to gossip with. We promote this service to the level of a first-class abstraction of a large-scale distributed system, similar to a name service being a first-class abstraction of a local-area system.

One important problem when implementing a peer sampling service in large dynamic networks is making sure that it is as scalable and robust as possible. We present a generic framework to implement a peer sampling service in a decentralized manner by constructing and maintaining *dynamic unstructured* overlays through gossiping membership information itself. Our framework generalizes existing approaches and makes it easy to discover new ones. We use this framework to empirically explore and compare several implementations of the peer sampling service. Through extensive simulation experiments we show that—although all protocols provide a good quality uniform random stream of peers to each node locally—traditional theoretical assumptions about the randomness of the unstructured overlays as a whole do not hold in any of the instances. We also show that different design decisions result in severe differences from the point of view of two crucial aspects: load balancing and fault tolerance. Our simulations are validated by means of a wide-area implementation.

2.1 Introduction

The popularity of gossip protocols stems from their ability to reliably pass information among a large set of interconnected nodes, even if the nodes regularly join and leave the system (either purposefully or on account of failures), and the underlying network suffers from broken or slow links.

In a gossip-based protocol, each node in the system periodically exchanges information with a subset of its peers. The choice of this subset is crucial to the wide dissemination of the gossip. Ideally, any given node should exchange information with peers that are selected following a uniform random sample of *all nodes* currently in the system [20, 22, 52–54]. This assumption made it possible to rigorously establish many desirable features of gossip-based protocols like scalability, reliability, and efficiency (see Chapter 1).

In practice, enforcing this assumption would require to develop applications where each node may be assumed to know every other node in the system [53, 55, 56]. However, providing each node with a complete membership table from which a random sample can

be drawn, is unrealistic in a large-scale dynamic system, for maintaining such tables in the presence of joining and leaving nodes (referred to as *churn*) incurs considerable synchronization costs. In particular, measurement studies on various peer-to-peer networks indicate that an individual node may often be connected in the order of only a few minutes to an hour (see, e.g. [57, 58]).

Clearly, decentralized schemes to maintain membership information are crucial to the deployment of gossip-based protocols. This chapter factors out the very abstraction of a *peer sampling service* and presents a generic, yet simple, gossip-based framework to implement it.

The peer sampling service is singled-out from the application using it and, abstractly speaking, the same service can be used in different settings: information dissemination [20,59], aggregation [8,10,34,60], load balancing [61], and network management [36]. The service is promoted as a first class abstraction of a large-scale distributed system. In a sense, it plays the role of a naming service in a traditional LAN-oriented distributed system as it provides each node with other nodes to interact with.

The basic general principle underlying the framework we propose to implement the peer sampling service, is itself based on a gossip paradigm. In short, every node (1) maintains a relatively small local membership table that provides a *partial view* on the complete set of nodes and (2) periodically refreshes the table using a gossiping procedure. The framework is generic and can be used to instantiate known [5, 62, 63] and novel gossip-based membership implementations. In fact, our framework captures many possible variants of gossip-based membership dissemination. These variants mainly differ in the way the membership table is updated at a given node after the exchange of tables in a gossip cycle. We use this framework to experimentally evaluate various implementations and identify key design parameters in practical settings. Our experimentation covers both extensive simulations and emulations on a wide-area cluster.

We consider many dimensions when identifying qualitative differences between the variants we examine. These dimensions include the randomness of selecting a peer as perceived by a single node, the accuracy of the current membership view, the distribution of the load incurred on each node, as well as the robustness in the presence of failures and churn.

Maybe not surprisingly, we show that communication should rather be bidirectional: it should follow the push-pull model. Adhering to a push-only or pull-only approach can easily lead to (irrecoverable) partitioning of the set of nodes. Another finding is that robustness against failing nodes or churn can be enhanced if old table entries are dropped when exchanging membership information.

However, as we shall also see, no single implementation outperforms the others along all dimensions. In this study we identify these tradeoffs when selecting an implementation of the peer sampling service for a given application. For example, to achieve good load balancing, table entries should rather be *swapped* between two peers. However, this strategy is less robust against failures and churn than non-swapping ones.

The chapter is organized as follows. Section 2.2 presents the interface and generic implementation of our peer sampling service. Section 2.3 characterizes *local* randomness: that is, the randomness of the samples as seen by a fixed participating node. In Section 2.4 we analyze *global* randomness in a graph-theoretic framework. Robustness to failures and churn is discussed in Section 2.5. The simulations are validated through a wide-area experimentation described in Section 2.6. Sections 2.7, 2.8 and 2.9 present the discussion, related work and conclusions, respectively.

2.2 Peer-Sampling Service

The peer sampling service is implemented over a set of nodes (a group) wishing to execute one or more protocols that require random samples from the group. The task of the service is to provide a participating node with a random subset of peers from the group.

2.2.1 API

The API of the peer sampling service simply consists of two methods: INIT and SELECTPEER. It would be technically straightforward to provide a framework for a multiple-application interface and architecture. For a better focus and simplicity of notations we assume, however, that there is only one application. The specification of these methods is as follows.

INIT() Initializes the service on a given node if this has not been done before. The actual initialization procedure is implementation dependent.

SELECTPEER() Returns a peer address if the group contains more than one node. The returned address is a sample drawn from the group. Ideally, this sample should be an independent unbiased random sample. The exact characteristics of this sample (e.g., its randomness or correlation in time and with other peers) is affected by the implementation.

Our focus is to give accurate information about the behavior of the SELECTPEER() method in the case of a class of gossip-based implementations. Applications requiring more than one peer simply invoke this method repeatedly.

Note that we do not define a STOP method. In other words, graceful leaves are handled as crashes. The reason is to ease the burden on applications by delegating the responsibility of removing inactive nodes to the service layer.

2.2.2 Generic Protocol Description

We consider a set of nodes connected in a network. A node has an address that is needed for sending a message to that node. Each node maintains a membership table representing its (partial) knowledge of the global membership. Traditionally, if this knowledge is complete, the table is called the *global view* or simply the *view*. However, in our case each node knows only a limited subset of the system, so the table is consequently called a *partial view*. The partial view is a list of c *node descriptors*. Parameter c represents the size of the list and is the same for all nodes.

A node descriptor contains a network address (such as an IP address) and an *age* that represents the freshness of the given node descriptor. The partial view is a list data structure, and accordingly, the usual list operations are defined on it. Most importantly, this means that the order of elements in the view is not changed unless some specific method (for example, SHUFFLE, which randomly reorders the list elements) explicitly changes it. The protocol also ensures that there is at most one descriptor for the same address in every view.

The purpose of the gossiping algorithm, executed periodically on each node and resulting in two peers exchanging their membership information, is to make sure that the partial views contain descriptors of a continuously changing random subset of the nodes

Algorithm 7 The skeleton of a gossip-based peer sampling service.

```
 1: loop
 2:    wait(Δ)
 3:    p ← selectGPSPeer()
 4:    sendPush( p, toSend() )
 5:    view.increaseAge()
 6:
 7: procedure ONPUSH(m)
 8:    if pull then
 9:       sendPull( m.sender, toSend() )
10:    onPull(m)
11:
12: procedure ONPULL(m)
13:    update(m.buffer,c,H,S)
14:    view.increaseAge()
15: procedure UPDATE(buffer,c,H,S)
16:    view.append(buffer)
17:    view.removeDuplicates()
18:    view.removeOldItems(min(H,view.size-c))
19:    view.removeHead(min(S,view.size-c))
20:    view.removeAtRandom(view.size-c)
21:
22: procedure TOSEND
23:    buffer ← ((MyAddress,0))
24:    view.shuffle()
25:    move oldest H items to end of view
26:    buffer.append(view.head(c/2 − 1))
27:    return buffer
```

and (in the presence of failure and joining and leaving nodes) to make sure the partial views reflect the dynamics of the system. We assume that each node executes the same protocol of which the skeleton is shown in Algorithm 7.

Note that the algorithm is an instantiation of the generic scheme in Algorithm 6, only it is slightly simpler because we do not support the pure pull variant, as explained in Section 2.2.3. As in Chapter 1, we define a cycle to be a time interval of Δ time units where Δ is the parameter of the protocol. During a cycle, each node initiates one view exchange.

Three globally known system-wide parameters are used in this algorithm: parameters c (the size of the partial view of each node), H and S. For the sake of clarity, we leave the details of the meaning and impact of H and S until the end of this section.

In the active thread, first a peer node is selected to exchange membership information with. This selection is implemented by the method SELECTGPSPEER that returns the address of a *live* node. This method *should not be confused* with the API method SELECTPEER. Although it serves a similar purpose, method SELECTGPSPEER is internal to the peer sampling implementation, and it itself is a parameter of the generic protocol. We discuss the possible implementations of SELECTGPSPEER in Section 2.2.3.

Subsequently, a push message is sent. The list of descriptors to be sent is prepared by method TOSEND. There, a buffer is initialized with a fresh descriptor of the node running the thread. Then, $c/2-1$ elements are appended to the buffer. The implementation ensures that these elements are selected randomly from the view without replacement, giving the oldest H elements (as defined by the age stored in the descriptors) a lower priority to be included (they are sampled only if there are not enough younger elements). As a side-effect of shuffling the view to select the $c/2 - 1$ random elements without replacement, the view itself will also have exactly those elements as first items (i.e., in the list head) that are being sent in the buffer. This fact will play a key role in the interpretation of parameter S as we explain later. Parameter H is guaranteed to be less than or equal to $c/2$. The buffer created this way is sent to the selected peer.

When a push message arrives from a peer node (in method ONPUSH) then if the boolean parameter PULL is true then a pull message is sent also prepared by TOSEND. When any message arrives from a peer node (in method ONPUSH or ONPULL) the received

buffer is passed to method UPDATE, which creates the new view based on the listed parameters, and the current view, making sure the size of the new view does not decrease and is at most c. After appending the received buffer to the view, method UPDATE keeps only the freshest entry for each address, eliminating duplicate entries. After this operation, there is at most one descriptor for each address. At this point, the size of the view is guaranteed to be at least the original size, since in the original view each address was included also at most once. Subsequently, the method performs a number of removal steps to decrease the size of the view to c. The parameters of the removal methods are calculated in such a way that the view size never drops below c. First, the oldest items are removed, as defined by their age, and parameter H. The name H comes from *healing*, that is, this parameter defines how aggressive the protocol should be when it comes to removing links that potentially point to faulty nodes (dead links). Note that in this way self-healing is implemented without actually checking if a node is alive or not. If a node is not alive, then its descriptors will never get refreshed (and thus become old), and therefore sooner or later they will get removed. The larger H is, the sooner older items will be removed from views.

After removing the oldest items, the S first items are removed from the view. Recall that it is exactly these items that were sent to the peer previously. As a result, parameter S controls the priority that is given to the addresses received from the peer. If S is high, then the received items will have a higher probability to be included in the new view. Since the same algorithm is run on the receiver side, this mechanism in fact controls the number of items that are *swapped* between the two peers, hence the name S for the parameter. This parameter controls the diversity of the union of the two new views (on the passive and active side). If S is low then both parties will keep many of their exchanged elements, effectively increasing the similarity between the two respective views. As a result, more unique addresses will be removed from the system. In contrast, if S is high, then the number of unique addresses that are lost from both views is lower. The last step removes random items to reduce the size of the view back to c.

This framework captures the essential behavior of many existing gossip membership protocols (although exact matches often require small changes). As such, the framework serves two purposes: (1) we can use it to compare and evaluate a wide range of different gossip membership protocols by changing parameter values, and (2) it can serve as a unifying implementation for a large class of protocols. As a next step, we will explore the design space of our framework, forming the basis for an extensive protocol comparison.

2.2.3 Design Space

In this section we describe a set of specific instances of our generic protocol by specifying the values of the key parameters. These instances will be analyzed in the rest of the chapter.

Peer Selection

As described before, peer selection is implemented by SELECTGPSPEER that returns the address of a *live* node as found in the caller's current view. In this study, we consider the following *peer selection* policies:

rand	Uniform randomly select an available node from the view
tail	Select the node with the *highest* age

Note that the third logical possibility of selecting the node with the *lowest* age is not included since this choice is not relevant. It is immediately clear from simply considering the protocol scheme that node descriptors with a low age refer to neighbors that have a view that is strongly correlated with the node's own view. More specifically, the node descriptor with the lowest age always refers exactly to the last neighbor the node communicated with. As a result, contacting this node offers little possibility to update the view with unknown entries, so the resulting overlay will be very static. Our preliminary experiments fully confirm this simple reasoning. Since the goal of peer sampling is to provide uncorrelated random peers continuously, it makes no sense to consider any policies with a bias towards low age, and thus protocols that follow such a policy.

View propagation

Once a peer has been chosen, the peers may exchange information in various ways. We consider the following two *view propagation* policies:

push	The node sends descriptors to the selected peer
pushpull	The node and selected peer exchange descriptors

Like in the case of the view selection policies, one logical possibility: the *pull* strategy, is omitted. It is easy to see that the pull strategy cannot possibly provide satisfactory service. The most important flaw of the pull strategy is that a node cannot inject information about itself, except only when explicitly asked by another node. This means that if a node loses all its incoming connections (which might happen spontaneously even without any failures, and which is rather common as we shall see) there is no possibility to reconnect to the network.

View selection

The parameters that determine how view selection is performed are H, the self-healing parameter, and S, the swap parameter. Let us first note some properties of these parameters. First, assuming that c is even, all values of H for which $H > c/2$ are equivalent to $H = c/2$, because the protocol never decreases the view size to under c. For the same reason, all values of S for which $S > c/2 - H$ are equivalent to $S = c/2 - H$. Furthermore, the last, random removal step of the view selection algorithm is executed only if $S < c/2 - H$. Keeping these in mind, we have a "triangle" of protocols with H ranging from 0 to $c/2$, and with S ranging from 0 to $c/2 - H$. In our analysis we will look at this triangle at different resolutions, depending on the scenarios in question. As a minimum, we will consider the three vertices of the triangle defined as follows.

blind	$H = 0, S = 0$	Keep blindly a random subset
healer	$H = c/2$	Keep the freshest entries
swapper	$H = 0, S = c/2$	Minimize loss of information

We must note here that even in the case of SWAPPER, only at most $c/2 - 1$ descriptors can be swapped, because the first element of the received buffer of length $c/2$ is always a fresh descriptor of the sender node. This fresh descriptor is always added to the view of the recipient node if $H + S = c/2$, that is, when no random elements are removed. This detail is very important as it is the only way fresh information can enter the system.

Algorithm 8 Newscast

```
 1: loop
 2:   wait(Δ)
 3:   p ← selectGPSPeer()
 4:   sendPush( p, toSend() )
 5:   view.increaseAge()
 6:
 7: procedure ONPUSH(m)
 8:   sendPull( m.sender, toSend() )
 9:   onPull(m)
10:
11: procedure ONPULL(m)
12:   update(m.buffer,c)
13:   view.increaseAge()
14: procedure UPDATE(buffer,c)
15:   view.append(buffer)
16:   view.removeDuplicates()
17:   view.removeOldItems(view.size-c)
18:
19: procedure TOSEND
20:   buffer ← ((MyAddress,0))
21:   buffer.append(view)
22:   return buffer
```

2.2.4 Known Protocols as Instantiations of the Model

The framework captures a number of protocols that were published previously. Here we briefly describe each in turn. The first is LPBCAST (lightweight probabilistic broadcast) [62]. The original publication describes a complete system for implementing a publish-subscribe service. A part of that system is a membership management layer, that is implemented essentially as the push variant of protocol BLIND with peer selection RAND. The second protocol we cover is called CYCLON [63]. Apart from minor differences, Cyclon is equivalent to push-pull SWAPPER with RAND as peer selection.

Finally, we discuss NEWSCAST in more detail. The NEWSCAST protocol was published originally as a technical report [4] that was later reprinted as a book chapter [5]. The first version of the NEWSCAST protocol also included an aggregation protocol, and later on the membership service was factored out in several subsequent publications when it became clear that it is useful for a wide range of other applications as well. The pseudocode of NEWSCAST is shown in Algorithm 8. The difference from Algorithm 7 is that NEWSCAST is always push-pull, the entire view is transferred (that is, c elements and not $c/2$ elements), and the freshest c elements are kept from the union of the views by both nodes that participate in an exchange. This results in an increased aggressiveness in removing old values even w.r.t. HEALER. Method SELECTGPSPEER simply returns a random element from the current view (that is, we use the RAND peer selection variant).

2.2.5 Implementation

We now describe a possible implementation of the peer sampling service API based on the framework presented in Section 2.2.2. We assume that the service forms a layer between the application and the unstructured overlay network.

Initialization

Method INIT() will cause the service to register itself with the gossiping protocol instance that maintains the overlay network. From that point, the service will be notified by this instance whenever the actual view is updated.

Sampling

As an answer to the SELECTPEER call, the service returns an element from the *current* view. To increase the randomness of the returned peers, the service makes a best effort not to return the same element twice during the period while the given element is in the view: this would introduce an obvious bias that would damage the quality of the service. To achieve this, the service maintains a queue of elements that are currently in the view but have not been returned yet. Method SELECTPEER returns the first element from the queue and subsequently it removes this element from the queue. When the service receives a notification on a view update, it removes those elements from the queue that are no longer in the current view, and appends the new elements that were not included in the previous view. If the queue becomes empty, the service falls back on returning random samples from the current view. In this case the service can set a warning flag that can be read by applications to indicate that the quality of the returned samples is no longer reliable.

In the following sections, we analyze the behavior of our framework in order to gradually come to various optimal settings of the parameters. Anticipating our discussion in Section 2.7, we will show that there are some parameter values that never lead to good results (such as selecting a peer from a fresh node descriptor). However, we will also show that no single combination of parameter values is always best and that, instead, tradeoffs need to be made.

2.3 Local Randomness

Ideally, a peer-sampling service should return a series of unbiased independent random samples from the current group of peers. The assumption of such randomness has indeed led to rigorously establish many desirable features of gossip-based protocols like scalability, reliability, and efficiency [21].

When evaluating the quality of a particular implementation of the service, one faces the methodological problem of characterizing randomness. In this section we consider a fixed node and analyze the series of samples generated at that particular node.

There are essentially two ways of capturing randomness. The first approach is based on the notion of Kolmogorov complexity [64]. Roughly speaking, this approach considers as random any series that cannot be compressed. Pseudo random number generators are automatically excluded by this definition, since any generator, along with a random seed, is a compressed representation of a series of any length. Sometimes it can be proven that a series *can* be compressed, but in the general case, the approach is not practical to test randomness due to the difficulty of proving that a series *cannot* be compressed.

The second, more practical approach assumes that a series is random if any statistic computed over the series matches the theoretical value of the same statistic under the assumption of randomness. The theoretical value is computed in the framework of probability theory. This approach is essentially empirical, because it can never be mathematically proven that a given series is random. In fact, good pseudo random number generators pass most of the randomness tests that belong to this category.

Following the statistical approach, we view the peer-sampling service (as seen by a fixed node) as a random number generator, and we apply the same traditional methodology that is used for testing random number generators. We test our implementations with the "diehard battery of randomness tests" [65], the *de facto* standard in the field.

2.3.1 Experimental Settings

We have experimented our protocols using the PEERSIM simulator [66]. All the simulation results in this chapter were obtained using this implementation.

The DIEHARD test suite requires as input a considerable number of 32-bit integers: the most expensive test needs $6 \cdot 10^7$ of them. To be able to generate this input, we assume that all nodes in the network are numbered from 0 to N. Node N executes the peer-sampling service, obtaining one number between 0 and $N-1$ each time it calls the service, thereby generating a sequence of integers. If N is of the form $N = 2^n + 1$, then the bits of the generated numbers form an unbiased random bit stream, provided the peer-sampling service returns random samples.

Due to the enormous cost of producing a large number of samples, we restricted the set of implementations of the view construction procedure to the three extreme points: BLIND, HEALER and SHUFFLER. Peer selection was fixed to be TAIL and PUSHPULL was fixed as the communication model. Furthermore, the network size was fixed to be $2^{10} + 1 = 1025$, and the view size was $c = 20$. These settings allowed us to complete $2 \cdot 10^7$ cycles for all the three protocol implementations. In each case, node N generated four samples in each cycle, thereby generating four 10-bit numbers. Ignoring two bits out of these ten, we generated one 32-bit integer for each cycle.

Experiments convey the following facts. No matter which two bits are ignored, it does not affect the results, so we consider this as a noncritical decision. Note that we could have generated 40 bits per cycle as well. However, since many tests in the DIEHARD suit do respect the 32-bit boundaries of the integers, we did not want to artificially diminish any potential periodic behavior in terms of the cycles.

2.3.2 Test Results

For a complete description of the tests in the DIEHARD benchmark we refer to [65]. In Table 2.1 we summarize the basic ideas behind each class of tests. In general, the three random number sequences pass all the tests, including the most difficult ones [67], with one exception. Before discussing the one exception in more detail, note that for two tests we did not have enough 32-bit integers, yet we could still apply them. The first case is the permutation test, which is concerned with the frequencies of the possible orderings of 5-tuples of subsequent random numbers. The test requires $5 \cdot 10^7$ 32-bit integers. However, we applied the test using the original 10-bit integers returned by the sampling service, and the random sequences passed. The reason is that ordering is not sensitive to the actual range of the values, as long as the range is not extremely small. The second case is the so called "gorilla" test, which is a strong instance of the class of the monkey tests [67]. It requires $6.7 \cdot 10^7$ 32-bit integers. In this case we concatenated the output of the three protocols and executed the test on this sequence, with a positive result. The intuitive reasoning behind this approach is that if any of the protocols produces a nonrandom pattern, then the entire sequence is supposed to fail the test, especially given that this test is claimed to be extremely difficult to pass.

Consider now the test that proved to be difficult to pass. This test was an instance of the class of binary matrix rank tests. In this instance, we take 6 consecutive 32-bit integers, and select the same (consecutive) 8 bits from each of the 6 integers forming a 6×8 binary matrix whose rank is determined. That rank can be from 0 to 6. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks smaller or equal to 4, and for ranks 5 and 6.

Birthday Spacings	The k-bit random numbers are interpreted as "birthdays" in a "year" of 2^k days. We take m birthdays and list the spacings between the consecutive birthdays. The statistic is the number of values that occur more than once in that list.
Greatest Comm. Divisor	We run Euclid's algorithm on consecutive pairs of random integers. The number of steps Euclid's algorithm needs to find the greatest common divisor (GCD) of these consecutive integers in the random series, and the GCD itself are the statistics used to test randomness.
Permutation	Tests the frequencies of the $5! = 120$ possible orderings of consecutive integers in the random stream.
Binary Matrix Rank	Tests the rank of binary matrices built from consecutive integers, interpreted as bit vectors.
Monkey	A set of tests for verifying the frequency of the occurrences of "words" interpreting the random series as the output of a monkey typing on a typewriter. The random number series is interpreted as a bit stream. The "letters" that form the words are given by consecutive groups of bits (e.g., for 2 bits there are 4 letters, etc).
Count the 1-s	A set of tests for verifying the number of 1-s in the bit stream.
Parking Lot	Numbers define locations for "cars." We continuously "park cars" and test the number of successful and unsuccessful attempts to place a car at the next location defined by the random stream. An attempt is unsuccessful if the location is already occupied (the two cars would overlap).
Minimum Distance	Integers are mapped to two or three dimensional coordinates and the minimal distance among thousands of consecutive points is used as a statistic.
Squeeze	After mapping the random integers to the interval $[0, 1)$, we test how many consecutive values have to be multiplied to get a value smaller than a given threshold. This number is used as a statistic.
Overlapping Sums	The sum of 100 consecutive values is used as a statistic.
Runs Up and Down	The frequencies of the lengths of monotonously decreasing or increasing sequences are tested.
Craps	200,000 games of craps are played and the number of throws and wins are counted. The random integers are mapped to the integers $1, \ldots, 6$ to model the dice.

Table 2.1: Summary of the basic idea behind the classes of tests in the DIEHARD test suite for random number generators. In all cases tests are run with several parameter settings. For a complete description we refer to [65].

When the selected byte coincides with the byte contributed by one call to the peer-sampling service (bits 0-7, 8-15, etc), protocols BLIND and SWAPPER fail the test. To better see why, consider the basic functioning of the rank test. In most of the cases, the rank of the matrix is 5 or 6. If it is 5, it typically means that the same 8-bit entry is copied twice into the matrix. Our implementation of the peer-sampling service explicitly ensures that the diversity of the returned elements is maximized in the short run (see Section 2.2.5). As a consequence, rank 6 occurs relatively more often than in the case of a true random sequence. Note that for many applications this property is actually an advantage. However, HEALER passes the test. The reason of this will become clearer later. As we will see, in the case of HEALER the view of a node changes faster and therefore the queue of the samples to be returned is frequently flushed, so the diversity-maximizing effect is less significant.

The picture changes if we consider only every 4th sample in the random sequence generated by the protocols. In that case, BLIND and SWAPPER pass the test, but HEALER fails. In this case, the reason of the failure of HEALER is exactly the opposite: there are relatively too many repetitions in the sequence. Taking only every 8th sample, all protocols pass the test.

Finally, note that even in the case of "failures," the numeric deviation from random behavior is rather small. The expected occurrences of ranks of $\leq$4, 5, and 6 are 0.94%, 21.74%, and 77.31%, respectively. In the first type of failure, when there are too many occurrences of rank 6, a typical failed test gives percentages 0.88%, 21.36%, and 77.68%. When ranks are too small, a typical failure is, for example, 1.05%, 21.89%, and 77.06%.

2.3.3 Conclusions

The results of the randomness tests suggest that the stream of nodes returned by the peer-sampling service is close to uniform random for all the protocol instances examined. Given that some widely used pseudo-random number generators fail at least some of these tests, this is a highly encouraging result regarding the quality of the randomness provided by this class of sampling protocols.

Based on these experiments we cannot, however, conclude on global randomness of the resulting graphs. Local randomness, evaluated from a peer's point of view is important, however, in a complex large-scale distributed system, where the stream of random nodes returned by the nodes might have complicated correlations, merely looking at local behavior does not reveal some key characteristics such as load balancing (existence of bottlenecks) and fault tolerance. In Section 2.4 we present a detailed analysis of the global properties of our protocols.

2.4 Global Randomness

In Section 2.3 we have seen that from a local point of view all implementations produce good quality random samples. However, statistical tests for randomness and independence tend to hide important *structural* properties of the system *as a whole*. To capture these global correlations, in this section we switch to a graph theoretical framework. To translate the problem into a graph theoretical language, we consider the *communication topology* or *overlay topology* defined by the set of nodes and their views (recall that SELECTPEER() returns samples from the view). In this framework the directed edges of the communication graph are defined as follows. If node a stores the descriptor of node b

in its view then there is a directed edge (a, b) from a to b. In the language of graphs, the question is how similar this overlay topology is to a random graph in which the descriptors in each view represent a uniform independent random sample of the whole node set?

In this section we consider graph-theoretic properties of the overlay graphs. An important example of such properties is the *degree distribution*. The indegree of node i is defined as the number of nodes that have i in their views. The outdegree is constant and equal to the view size c for all nodes. Degree distribution has many significant effects. Most importantly, degree distribution determines whether there are hot spots and bottlenecks from the point of view of communication costs. In other words, *load balancing* is determined by the degree distribution. It also has a direct relationship with reliability to different patterns of node failures [68], and has an effect on the exact way epidemics are spread [69]. Apart from the degree distribution we also analyze the clustering coefficient and average path length, as described and motivated in Section 2.4.2.

Our main goal is to explore the different design choices in the protocol space described in Section 2.2.2. More specifically, we want to assess the impact of the peer selection, view selection, and view propagation parameters. Accordingly, we chose to fix the network size to $N = 10^4$ and the maximal view size to $c = 30$. The results presented in this section were obtained using the PEERSIM simulation environment [66].

2.4.1 Properties of Degree Distribution

The first and most fundamental question is whether, for a particular protocol implementation, the communication graph has some stable properties, which it maintains during the execution of the protocol. In other words, we are interested in the *convergence behavior* of the protocols. We can expect several sorts of dynamics which include chaotic behavior, oscillations, and convergence. In case of convergence the resulting state may or may not depend on the initial configuration of the system. In the case of overlay networks we obviously prefer to have convergence towards a state that is independent of the initial configuration. This property is called *self-organization*. In our case it is essential that in a wide range of scenarios the protocol instances should automatically produce consistent and predictable behavior. Section 2.4.1 examines this question.

A related question is whether there is convergence and what kind of communication graph a protocol instance converges to. In particular, as mentioned earlier, we are interested in what sense overlay topologies deviate from certain random graph models. We discuss this issue in Section 2.4.1.

Finally, we are interested in looking at *local dynamic* properties along with *globally stable* degree distributions. That is, it is possible that while the overall degree distribution and its global properties such as maximum, variance, average, etc., do not change, the degree of the individual nodes does. This is preferable because in this case even if there are always bottlenecks in the network, the bottleneck will not be the same node all the time which greatly increases robustness and improves load balancing. Section 2.4.1 is concerned with these questions.

Convergence

We now present experimental results that illustrate the convergence properties of the protocols in three different bootstrapping scenarios:

protocol	partitioned runs	average number of clusters	average largest cluster
(rand,healer,push)	100%	22.28	9124.48
(rand,swapper,push)	0%	n.a.	n.a.
(rand,blind,push)	18%	2.06	9851.11
(tail,healer,push)	29%	2.17	9945.21
(tail,swapper,push)	97%	4.07	9808.04
(tail,blind,push)	10%	2.00	9936.20

Table 2.2: Partitioning of the push protocols in the growing overlay scenario. Data corresponds to cycle 300. Cluster statistics are over the partitioned runs only.

Growing In this scenario, the overlay network initially contains only one node. At the beginning of each cycle, 500 new nodes are added to the network until the maximal size is reached in cycle 20. The view of these nodes is initialized with only a single node descriptor, which belongs to the oldest, initial node. This scenario is the most pessimistic one for bootstrapping the overlays. It would be straightforward to improve it by using more contact nodes, which can come from a fixed list or which can be obtained using inexpensive local random walks on the existing overlay. However, in our discussion we intentionally avoid such optimizations to allow a better focus on the core protocols and their differences.

Lattice In this scenario, the initial topology of the overlay is a ring lattice, a structured topology. We build the ring lattice as follows. The nodes are first connected into a ring in which each node has a descriptor in its view that belongs to its two neighbors in the ring. Subsequently, for each node, additional descriptors of the nearest nodes are added in the ring until the view is filled.

Random In this scenario the initial topology is defined as a random graph, in which the views of the nodes were initialized by a uniform random sample of the peer nodes.

As we focus on the dynamic properties of the protocols, we did not wish to average out interesting patterns, so in all cases the result of a single run is shown in the plots. Nevertheless, we ran all the scenarios 100 times to gain data on the stability of the protocols with respect to the connectivity of the overlay. Connectivity is a crucial feature, a minimal requirement for all applications. The results of these runs show that in all scenarios, every protocol under examination creates a connected overlay network in 100% of the runs (as observed in cycle 300). The only exceptions were detected during the growing overlay scenario. Table 2.2 shows the push protocols. With the pushpull scheme we have not observed any partitioning.

The push versions of the protocols perform very poorly in the growing scenario in general. Figure 2.1 illustrates the evolution of the maximal indegree. The maximal indegree belongs to the central contact node that is used to bootstrap the network. After growing is finished in cycle 20, the pushpull protocols almost instantly balance the degree distribution thereby removing the bottleneck. The push versions, however, get stuck in this unbalanced state.

This is not surprising, because when a new node joins the network and gets an initial contact node to start with, the only way it can get an updated view is if some other node

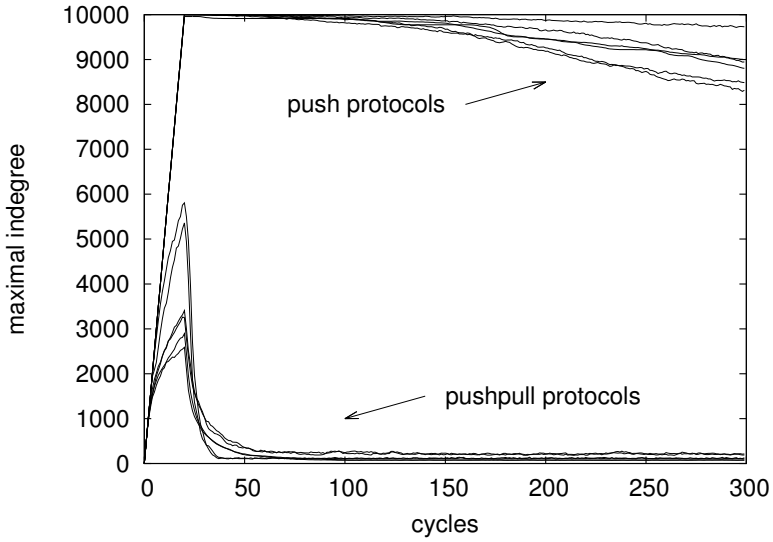


Figure 2.1: Evolution of maximal indegree in the growing scenario (recall that growing stops in cycle 20). The runs of the following protocols are shown: peer selection is either rand or tail, view selection is blind, healer or swapper, and view propagation is push or pushpull.

contacts it actively. This, however, is very unlikely. Because all new nodes have the same contact, the view at the contact node gets updated extremely frequently causing all the joining nodes to be quickly forgotten. A node has to push its own descriptor many times until some other node actually contacts it. This also means that if the network topology moves towards the shape of a star, then the push protocols have extreme difficulty balancing this degree-distribution state again towards a random one. *We conclude that this lack of adaptivity and robustness effectively renders push-only protocols useless.* In the following we therefore consider only the pushpull model.

Figure 2.2 illustrates the convergence of the pushpull protocols. Note that the average indegree is always the view size c. We can observe that in all scenarios the protocols quickly converge to the same value, even in the case of the growing scenario, in which the initial degree distribution is rather skewed. Other properties not directly related to degree distribution also show convergence, as discussed in Section 2.4.2.

Static Properties

In this section we examine the converged degree distributions generated by the different protocols. Figure 2.3 shows the converged standard deviation of the degree distribution. We observe that increasing both H and S results in a lower—and therefore more desirable—standard deviation. The reason is different for these two cases. With a large S, links to a node come to existence only in a very controlled way. Essentially, new incoming links to a node are created only when the node itself injects its own fresh node descriptor during communication. On the other hand, with a large H, the situation is the opposite. When a node injects a new descriptor about itself, this descriptor is (exponentially often) copied to other nodes for a few cycles. However, one or two cycles later all copies are removed because they are pushed out by new links (i.e., descriptors) injected in the meantime. So the effect that reduces variance is the short lifetime of the copies of

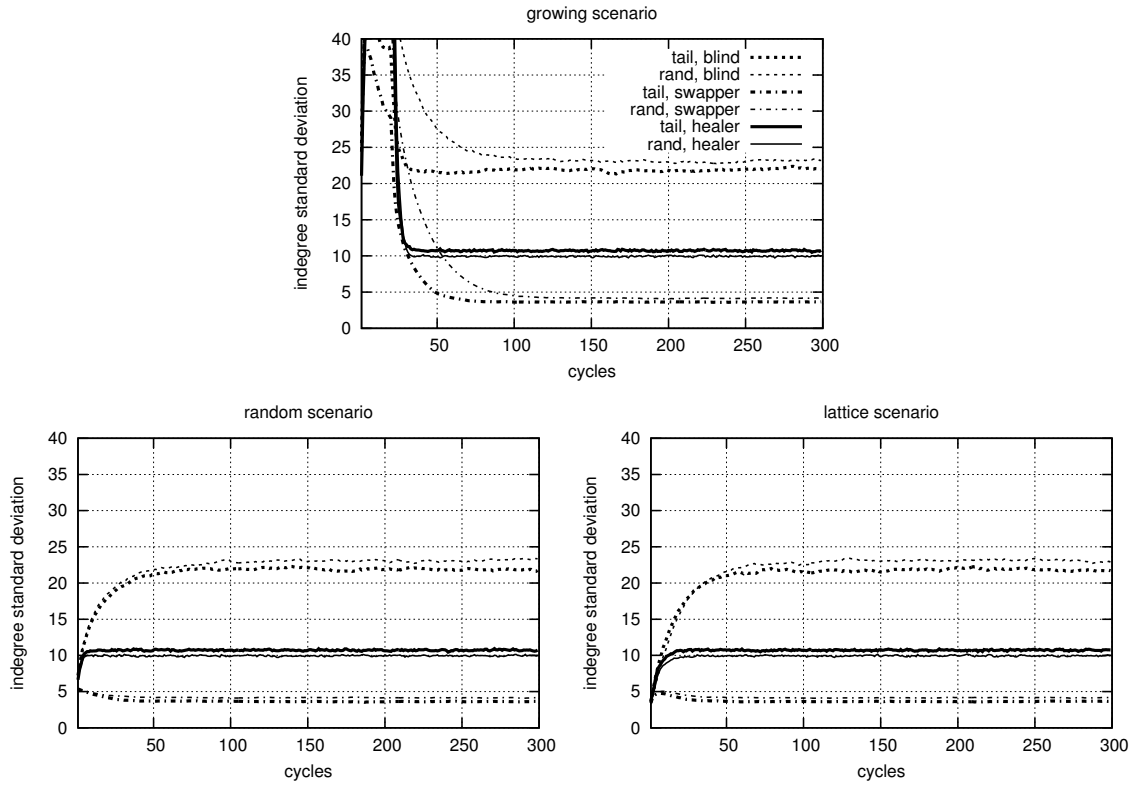


Figure 2.2: Evolution of standard deviation of indegree in all scenarios of pushpull protocols.

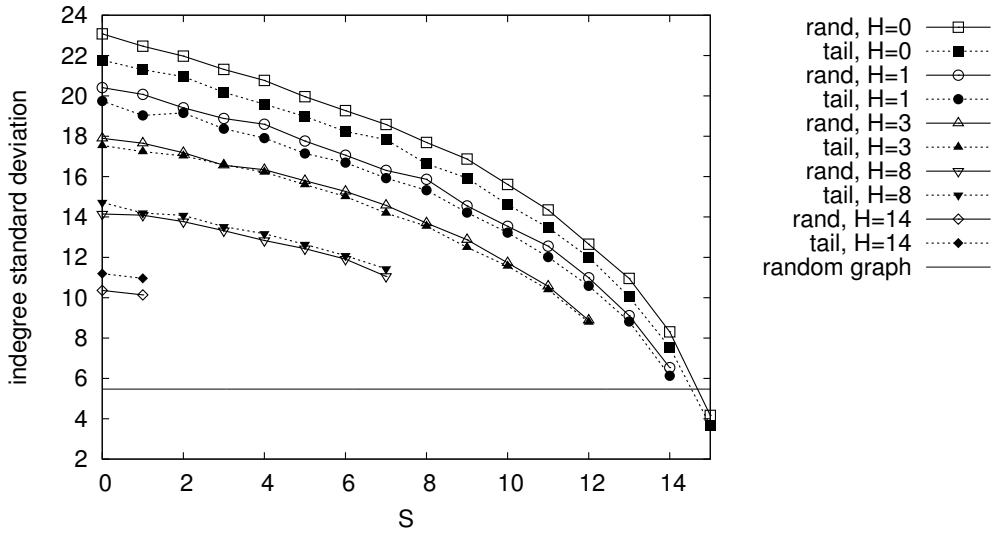


Figure 2.3: Converged values of indegree standard deviation.

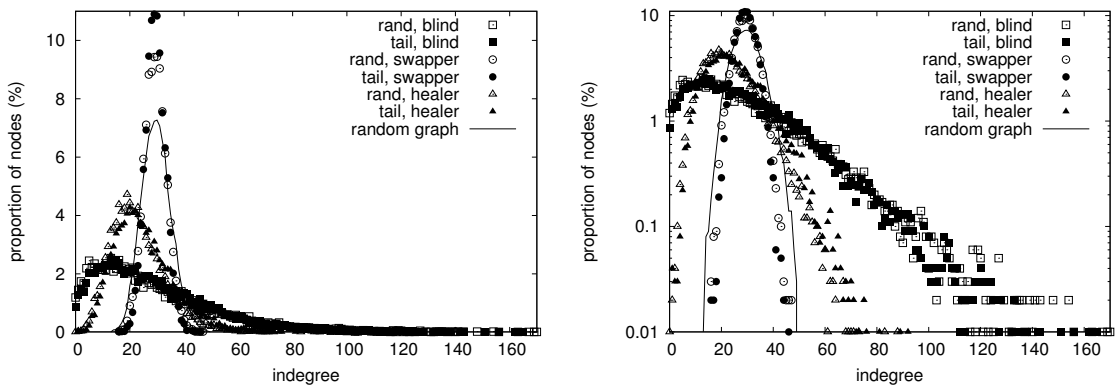


Figure 2.4: Converged indegree distributions on linear and logarithmic scales.

a given link.

Figure 2.4 shows the entire degree distribution for the three vertices of the design space triangle. We observe that the distribution of SWAPPER is narrower than that of the random graph, while BLIND has a rather heavy tail and also a large number of nodes with zero or very few nodes pointing to them, which is not desirable from the point of view of load balancing.

Dynamic Properties

Although the distribution itself does not change over time during the continuous execution of the protocols, the behavior of a single node still needs to be determined. More specifically, we are interested in whether a given fixed node has a variable indegree or whether the degree changes very slowly. The latter case would be undesirable because an unlucky node having above-average degree would continuously receive above-average traffic while others would receive less, which results in inefficient load balancing.

Figure 2.5 compares the degree distribution of a node over time, and the entire network at a fixed time point. The figure shows only the distribution for one node and only the random peer-selection protocols, but the same result holds for tail peer selection and for all the 100 other nodes we have observed. From the fact that these two distributions are very similar, we can conclude that all nodes take all possible values at some point in time, which indicates that the degree of a node is not static.

However, it is still interesting to characterize *how quickly* the degree changes, and whether this change is predictable or random. To this end, we present autocorrelation data of the degree time-series of fixed nodes in Figure 2.6. The band indicates a 99% confidence interval assuming the data is random. Only one node is shown, but all the 100 nodes we traced show very similar behavior. Let the series $d_1, \ldots d_K$ denote the indegree of a fixed node in consecutive cycles, and $\overline{d}$ the average of this series. The autocorrelation of the series $d_1, \ldots d_K$ for a given time lag k is defined as

$$r_k = \frac{\sum_{j=1}^{K-k}(d_j - \overline{d})(d_{j+k} - \overline{d})}{\sum_{j=1}^{K}(d_j - \overline{d})^2},$$

which expresses the correlation of pairs of degree values separated by k cycles.

We observe that in the case of HEALER it is impossible to make any prediction for a degree of a node 20 cycles later, knowing the current degree. However, for the rest of

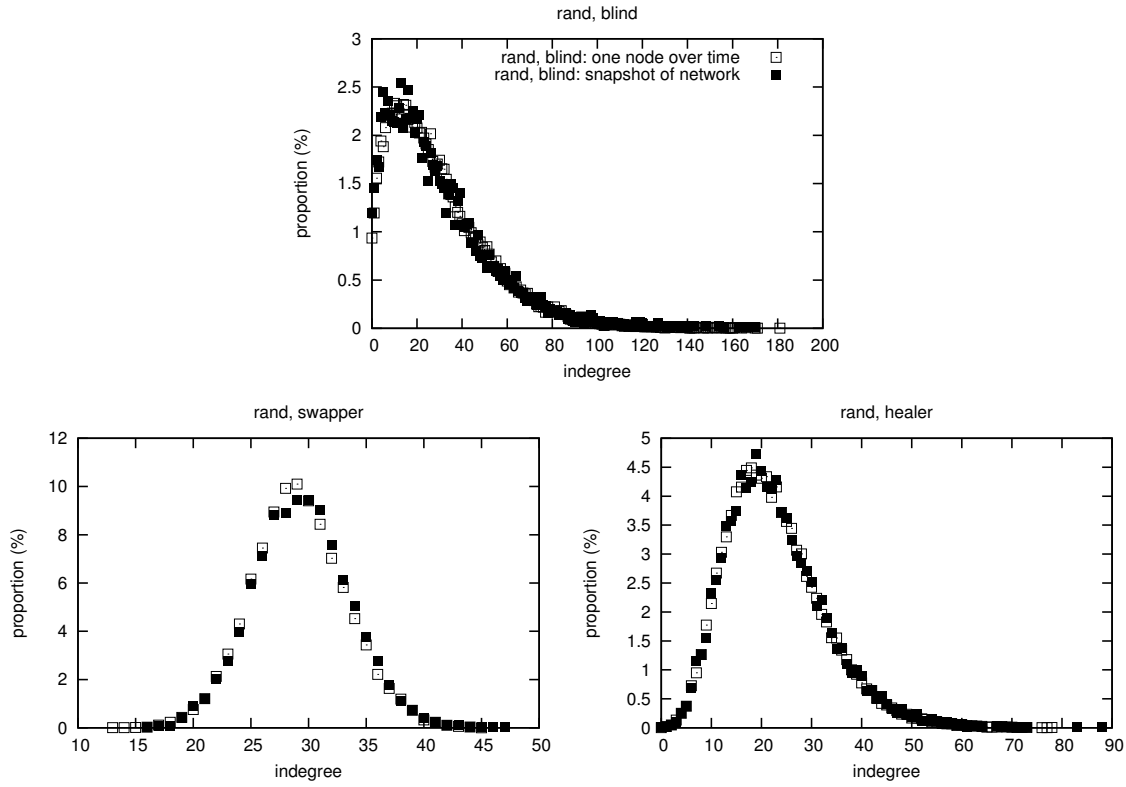


Figure 2.5: Comparison of the converged indegree distribution over the network at a fixed time point and the indegree distribution of a fixed node during an interval of 50,000 cycles. The vertical axis represents the proportion of nodes and cycles, respectively.

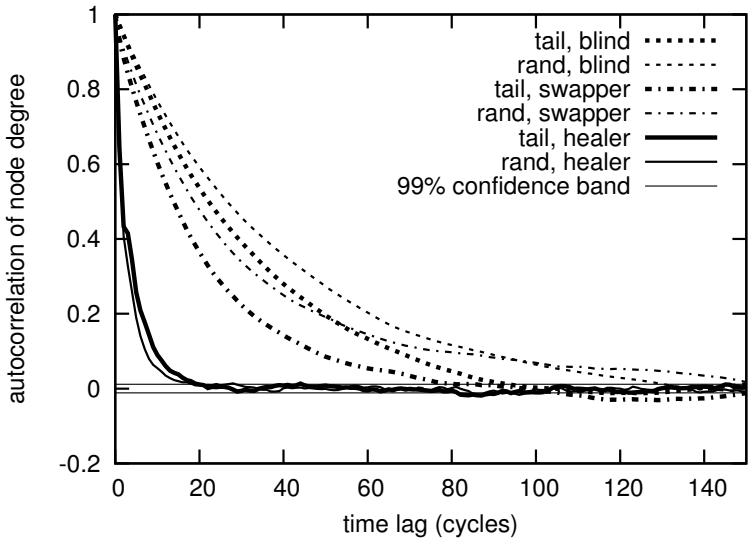


Figure 2.6: Autocorrelation of indegree of a fixed node over 50,000 cycles. Confidence band corresponds to the randomness assumption: a random series produces correlations within this band with 99% probability.

the protocols, the degree changes much slower, resulting in correlation in the distance of 80-100 cycles, which is not optimal from the point of view of load balancing.

2.4.2 Clustering and Path Lengths

Degree distribution is an important property of random graphs. However, there are other equally important characteristics of networks that are independent of degree distribution. In this section we consider the *average path length* and the *clustering coefficient* as two such characteristics. The clustering coefficient is defined over undirected graphs (see below). Therefore, we consider the undirected version of the overlay after removing the orientation of the edges.

Average path length

The shortest path length between node a and b is the minimal number of edges required to traverse in the graph in order to reach b from a. The average path length is the average of the shortest path lengths over all pairs of nodes in the graph. The motivation of looking at this property is that, in any information dissemination scenario, the shortest path length defines a lower bound on the time and costs of reaching a peer. For the sake of scalability a small average path length is essential. In Figure 2.7, especially in the growing and lattice scenarios, we verify that the path length converges rapidly. Figure 2.8 shows the converged values of the average path length for the design space triangle defined by H and S. We observe that all protocols result in a very low path length. Large S values are the closest to the random graph.

Clustering coefficient

The clustering coefficient of a node a is defined as the number of edges between the neighbors of a divided by the number of all possible edges between those neighbors. Intuitively, this coefficient indicates the extent to which the neighbors of a are also neighbors of each other. The clustering coefficient of a graph is the average of the clustering coefficients of its nodes, and always lies between 0 and 1. For a complete graph, it is 1, for a tree it is 0. The motivation for analyzing this property is that a high clustering coefficient has potentially damaging effects on both information dissemination (by increasing the number of redundant messages) and also on the self-healing capacity by weakening the connection of a cluster to the rest of the graph thereby increasing the probability of partitioning. Furthermore, it provides an interesting possibility to draw parallels with research on complex networks where clustering is an important research topic (e.g., in social networks) [70].

Like average path length, the clustering coefficient also converges (see Figure 2.7); Figure 2.8 shows the converged values. It is clear that clustering is controlled mainly by H. The largest values of H result in rather significant clustering, where the deviation from the random graph is large. The reason is that if H is large, then a large part of the views of any two communicating nodes will overlap right after communication, since both keep the same freshest entries. For the largest values of S, clustering is close to random. This is not surprising either because S controls exactly the diversity of views.

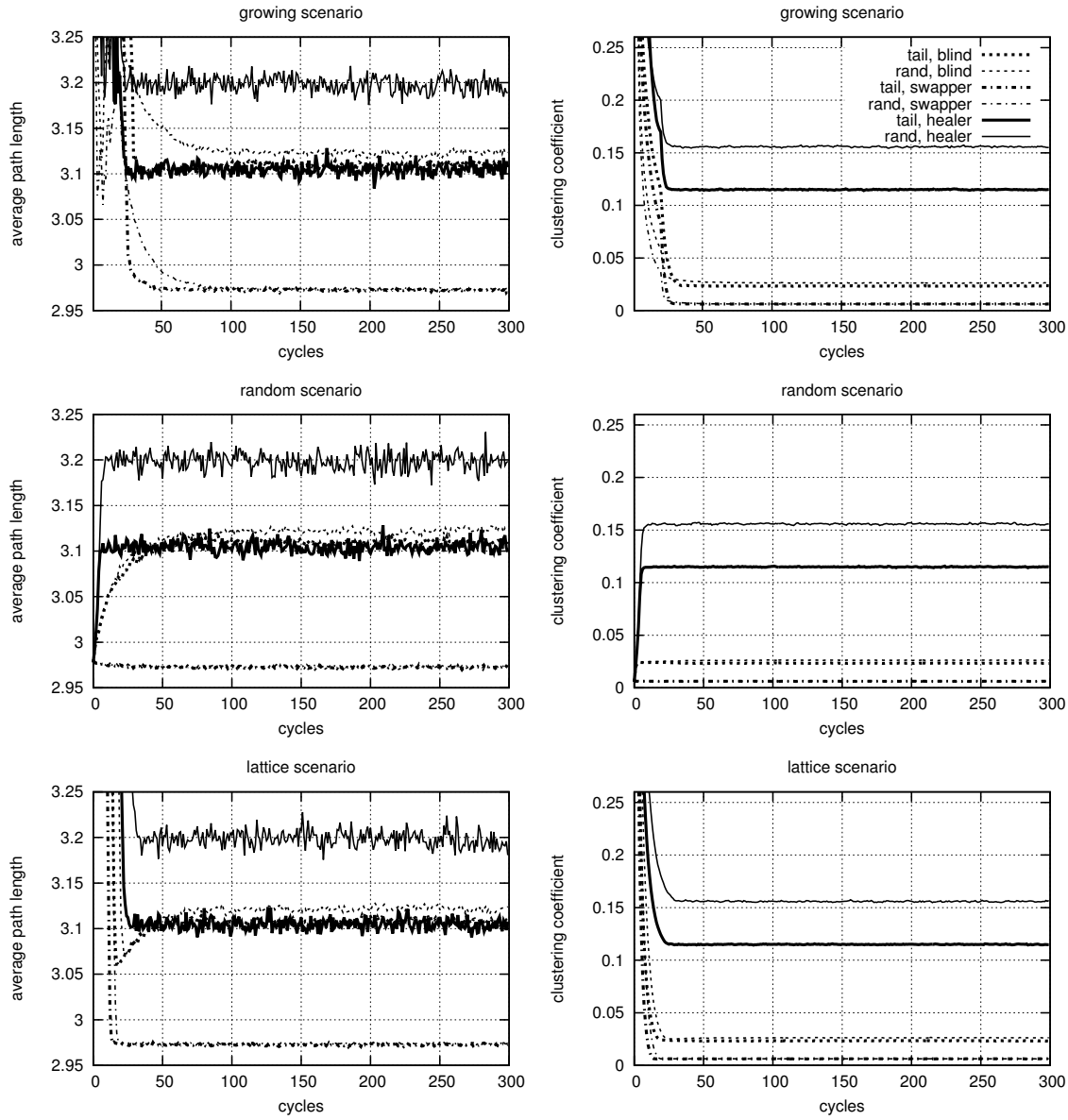


Figure 2.7: Evolution of the average path length and the clustering coefficient in all scenarios.

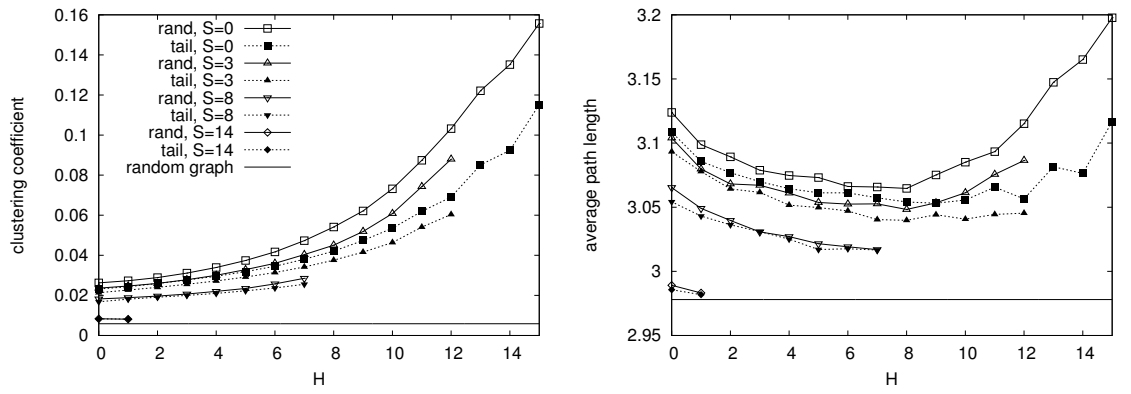


Figure 2.8: Converged values of clustering coefficient and average path length.

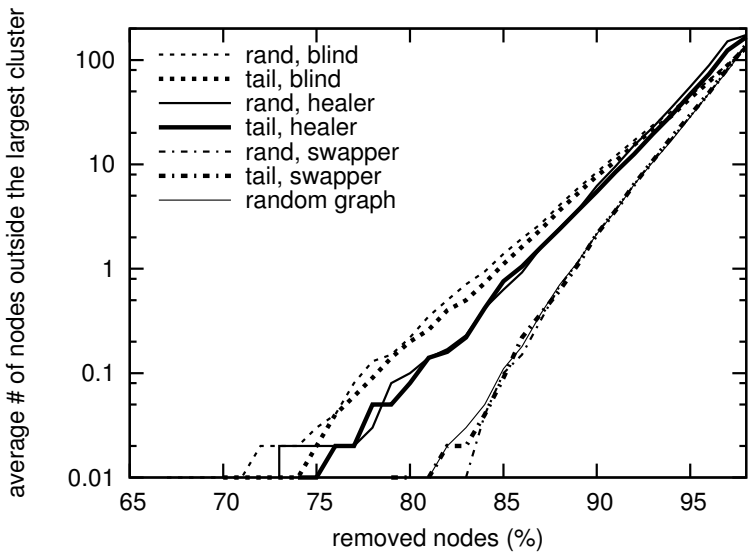


Figure 2.9: The number of nodes that do not belong to the largest connected cluster. The average of 100 experiments is shown. The random graph almost completely overlaps with the swapper protocols.

2.5 Fault Tolerance

In large-scale, dynamic, wide-area distributed systems it is essential that a protocol is capable of maintaining an acceptable quality of service under a wide range of severe failure scenarios. In this section we present simulation results on two classes of such scenarios: *catastrophic failure*, where a significant portion of the system fails at the same time, and heavy *churn*, where nodes join and leave the system continuously.

2.5.1 Catastrophic Failure

As in the case of the degree distribution, the response of the protocols to a massive failure has a static and a dynamic aspect. In the static setting we are interested in the self-healing capacity of the converged overlays to a (potentially massive) node failure, as a function of the number of failing nodes. Removing a large number of nodes will inevitably cause some serious structural changes in the overlay even if it otherwise remains connected. In the dynamic case we would like to learn to what extent the protocols can repair the overlay after a severe damage.

The effect of a massive node failure on connectivity is shown in Figure 2.9. In this setting the overlay in cycle 300 of the random initialization scenario was used as converged topology. From this topology, random nodes were removed and the connectivity of the remaining nodes was analyzed. In all of the $100 \times 6 = 600$ experiments performed we did not observe partitioning until removing 67% of the nodes. The figure depicts the number of the nodes outside the largest connected cluster. We observe consistent partitioning behavior over all protocol instances (with SWAPPER being particularly close to the random graph): even when partitioning occurs, most of the nodes form a single large connected cluster. Note that this phenomenon is well known for traditional random graphs [71].

In the dynamic scenario we made 50% of the nodes fail in cycle 300 of the random initialization scenario and we then continued running the protocols on the damaged over-

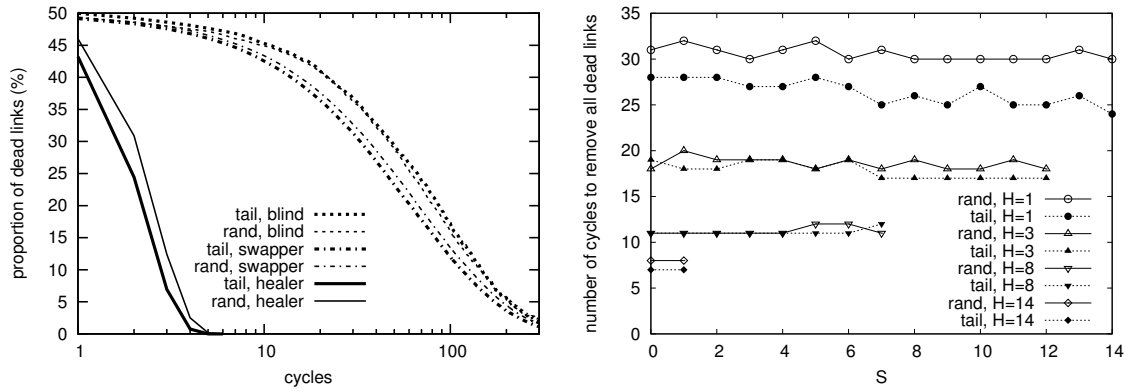


Figure 2.10: Removing dead links following the failure of 50% of the nodes in cycle 300.

lay. The damage is expressed by the fact that, on average, half of the view of each node consists of descriptors that belong to nodes that are no longer in the network. We call these descriptors dead links. Figure 2.10 shows how fast the protocols repair the overlay, that is, remove dead links from the views. Based on the static node failure experiment it was expected that the remaining 50% of the overlay is not partitioned and indeed, we did not observe partitioning with any of the protocols. Self-healing performance is fully controlled by the healing parameter H, with $H = 15$ resulting in fully repairing the network in as little as 5 cycles (not shown).

2.5.2 Churn

To examine the effect of churn, we define an artificial scenario in which a given proportion of the nodes crash and are subsequently replaced by new nodes in each cycle. This scenario is a *worst case* scenario because the new nodes are assumed to join the system for the first time, therefore they have no information whatsoever about the system (their view is initially empty) and the crashed nodes are assumed never to join the system again, so the links pointing to them will never become valid again. A more realistic trace-based scenario is also examined in Section 2.5.3 using the Gnutella trace described in [58].

We focus on two aspects: the churn rate, and the bootstrapping method. Churn rate defines the number of nodes that are replaced by new nodes in each cycle. We consider *realistic* churn rates (0.1% and 1%) and a *catastrophic* churn rate (30%). Since churn is defined in terms of cycles, in order to validate how realistic these settings are, we need to define the cycle length. With the very conservative setting of 10 seconds, which results in a very low load at each node, the trace described in [58] corresponds to 0.2% churn in each cycle. In this light, we consider 1% a comfortable upper bound of realistic churn, given also that the cycle length can easily be decreased as well to deal with even higher levels of churn.

We examine two bootstrapping methods. Both are rather unrealistic, but our goal here is not to suggest an optimal bootstrapping implementation, but to analyze our protocols under churn. The following two methods are suitable for this purpose because they represent two opposite ends of the design space:

Central We assume that there exists a server that is known by every joining node, and that is stable: it is never removed due to churn or other failures. This server participates in the gossip membership protocol as an ordinary node. The new nodes use the

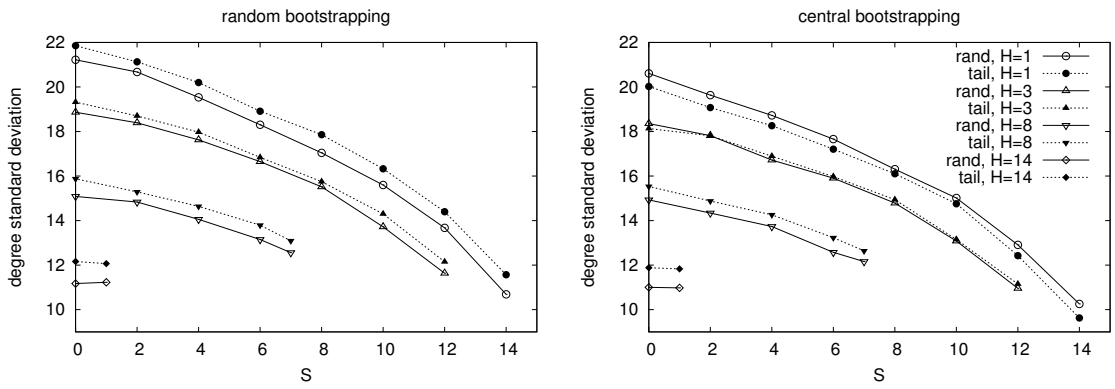


Figure 2.11: Standard deviation of node degree with churn rate 1%. Node degree is defined over the *undirected* version of the subgraph of *live* nodes. The $H = 0$ case is not comparable to the shown cases; due to reduced self-healing, nodes have much fewer live neighbors (see Figure 2.12) which causes relatively low variance.

server as their first contact. In other words, their view is initialized to contain the server.

Random An oracle gives each new node a random live peer from the network as its first contact.

Realistic implementations could use a combination of these two approaches, where one or more servers serve random contact peers, using the peer-sampling service itself. Any such implementation can reasonably be expected to result in a behavior in between the two extremes described above.

Simulation experiments were run initializing the network with random links and subsequently running the protocols under the given amount of churn until the observed properties reached a stable level (300 cycles). The experimental results reveal that for realistic churn rates (0.1% and 1%) all the protocols are robust to the bootstrapping method and the properties of the overlay are very close to those without churn. Figure 2.11 illustrates this by showing the standard deviation of the node degrees in both scenarios, for the higher churn rate 1%. Observe the close correspondence with Figure 2.3. The clustering coefficient and average path length show the same robustness to bootstrapping, and the observed values are almost identical to the case without churn (not shown).

Let us now consider the damage churn causes in the networks. First of all, for all protocols and scenarios the networks remain connected, even for $H = 0$. Still, a (low) number of dead links remain in the overlay. Figure 2.12 shows the average number of dead links in the views, again, only for the higher churn rate (1%). It is clear that the extent of the damage is fully controlled by the healing parameter H. Furthermore, it is clear that the protocols are robust to the bootstrapping scenario also in this case. If $H \geq 1$ then the *maximal* (not average) number of dead links in any view for the different protocol instances ranges from $5 - 13$ in the case of churn rate 1% and from $2 - 5$ for churn rate 0.1%, where the lowest value belongs to the highest H. If $H = 0$ then the number of dead links radically increases: it is at least 11 on average, and the maximal number of dead links ranges from 20-25 for the different settings. That is, in the presence of churn, it is essential for any implementation to set at least $H = 1$. We have already seen this effect in Section 2.5.1 concerning self-healing performance.

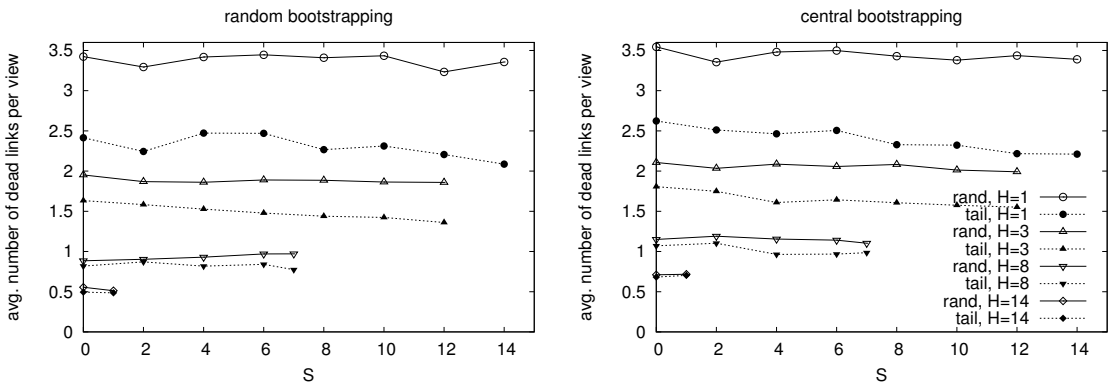


Figure 2.12: Average number of dead links in a view with churn rate 1%. The $H = 0$ case is not shown; it results in more than 11 dead links per view on average, for all settings.

Although the server participates in the overlay, the plots showing results under the central bootstrapping scenario were calculated ignoring the server, because its properties sharply differ from the rest of the network. In particular, it has a high indegree, because all new nodes will have a fresh link to the server, and that link will stay in the view of joining nodes for a few more cycles, possibly replicated in the meantime. Indeed, we observe that for 1% churn, $12\% - 28\%$ of the nodes have a link to the server at any time, depending on H and S. However, if we assume that the server can handle the traffic generated by joining nodes, a high indegree is noncritical. The expected number of incoming messages due to indegree d is d/c (where c is the view size), with a very low variance. This means that the generated traffic is of the same order of magnitude as the traffic generated by the joining nodes. We note again, however, that we do not consider this simplistic server-based solution a practical approach; we treat it only as a worst-case scenario to help us evaluate the protocols.

So far we have been discussing realistic churn rates. However, it is of academic interest to examine the behavior under *extremely* difficult scenarios, where the network suffers a catastrophic damage *in each cycle*. The catastrophic churn rate of 30% combines the effects of catastrophic failure (see Section 2.5.1) and churn.

Unlike with realistic churn rates, in this case the bootstrapping method has a strong effect on the performance of the protocols and therefore becomes the major design decision, although the parameters H and S still have a very strong effect as well. Consequently, we need to analyze the interaction of the gossip membership protocol and the bootstrapping method. In the case of the server-based solution, the overlay evolves into a ring-like structure, with a few shortcut links. The reason is that the view of the server is predominantly filled with entries of the newly joined nodes, since each time a new node contacts the server it also places a fresh entry about itself in the view of the server. These entries are served to the subsequently joining nodes, thus forming a linear structure. This ring-like structure is rather robust: it remains connected (even after removing the server) for all protocols with $H >= 8$. However, it has a slightly higher diameter than that of the random graph (approximately 20-30 hops). For HEALER the average number of dead links per view is still as low as 10 and 9 for random and tail peer selection, respectively.

The random scenario is rather different. In particular, we lose connectivity for all the protocols, however, for large values of H the largest connected cluster almost reaches the size of the network (see Figure 2.13). Besides, the structure of the overlay is also different. As Figure 2.13 shows, tail peer selection results in a slightly more unbalanced

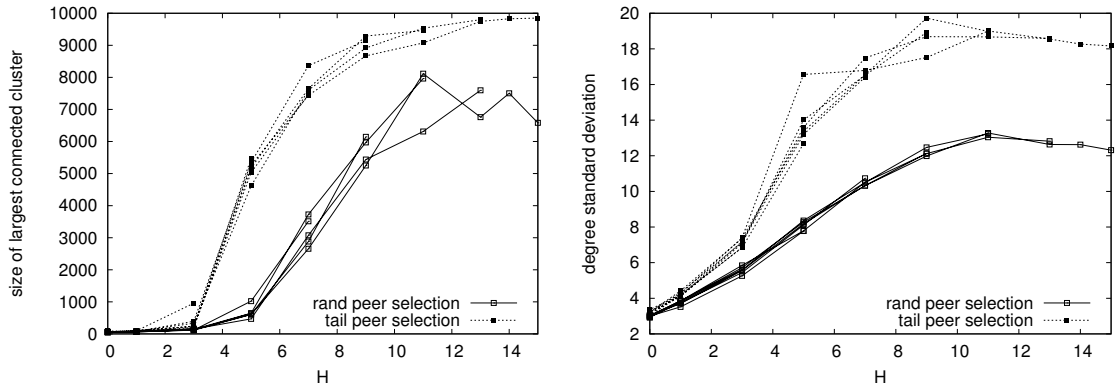


Figure 2.13: Size of largest connected cluster and degree standard deviation under catastrophic churn rate (30%), with the random bootstrapping method. Individual curves belong to different values of S but the measures depend only on H, so we do not need to differentiate between them. Connectivity and node degree are defined over the *undirected* version of the subgraph of *live* nodes.

degree distribution (note that the low deviation for low values of H is due to the low number of live nodes). The reason is that—also considering that tail peer selection picks the oldest *live* node—the nodes that stay in the overlay for somewhat longer will receive more incoming traffic because (due to the very high number of dead links in each view) they tend to be the oldest live node in most views they are in. For HEALER the average number of dead links per view is 11 and 9 for random and tail peer selection, respectively.

To summarize our findings: under realistic churn rates all the protocols perform very similarly to the case when there is no churn at all, independently of the bootstrapping method. Besides, some of the protocol instances, in particular, HEALER, can tolerate even catastrophic churn rates with a reasonable performance with both bootstrapping methods.

2.5.3 Trace-driven Churn Simulations

In Section 2.5.2 we analyzed our protocols under artificial churn scenarios. Here, we consider a realistic churn scenario using the, so called, *lifetime measurements* on Gnutella, carried out by Saroiu et al. [58]. These traces contain—among other information—the connection and disconnection times for a total of 17,125 nodes over a period of 60 hours. Throughout the trace, the number of connected nodes remains practically unchanged, in the order of 10^4 nodes.

We noticed a periodic pattern occurring every 404 seconds in the traces. In each 404-second interval, all connections and disconnections take place during the first 344 seconds, rendering the network static during the last 60 seconds. These recurring gaps would represent a positive bias for our churn simulations, as they periodically provide the overlay with some "breathing space" to process recent changes. However, these gaps are not realistic and are most probably an artifact of the logging mechanism. Therefore, we decided to eliminate them by linearly expanding each 344 second interval to cover the whole 404 seconds. Note that this transformation leaves the node uptimes practically unaltered.

We have taken the following two decisions with respect to the parameters in the experiments presented. First, peer selection is fixed to random. Section 2.5.2 showed that random is outperformed by tail peer selection in all cases. Therefore, random is a suitable

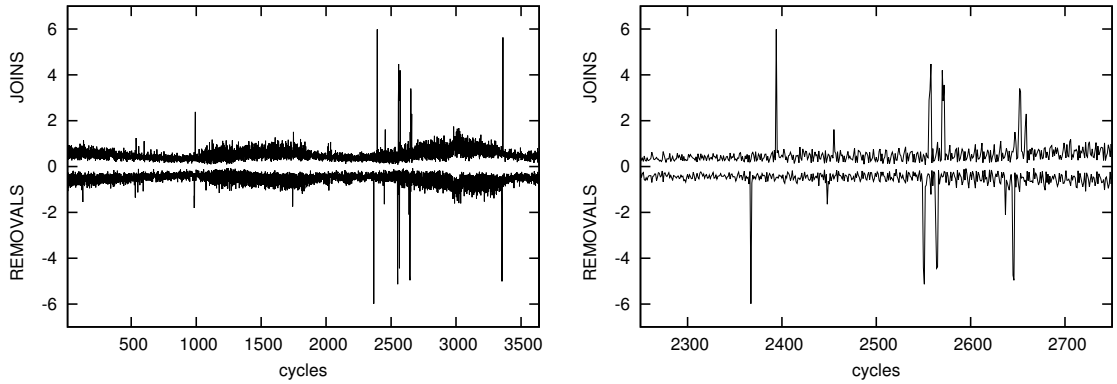


Figure 2.14: Churn in the Saroiu traces. Full time span of 3600 one minute cycles and zoomed in to cycles 2250 to 2750.

choice for this section as the worst case peer selection policy. Second, the swap parameter, S, is fixed to 0. Section 2.5.2 showed that $S = 0$ results in the highest (therefore worst) degree deviation, while it does not affect the number of dead links.

We apply two join methods: central and random, as defined in Section 2.5.2. The only difference is that a reconnecting node still remembers the links it previously had, some of which may be dead at reconnection time. This facilitates reconnection, but generally increases the total number of dead links.

The cycle length was chosen to be 1 minute. We anticipate that in reality the cycle length will be shorter, resulting in lower churn per cycle. The choice of a cycle length close to the upper end of realistic values is intentional, and is aimed at testing this specific gossip membership protocol under increased stress.

Figure 2.14 shows the node connections and disconnections as a percentage of the current network size. Connections are shown as positive points, whereas disconnections as negative. Although we ran the experiments for the whole trace, we focus on its most interesting part, namely cycles 2250 to 2750. Notice that at cycle 2367, around 450 nodes get disconnected at once and reconnect altogether 27 minutes later, at cycle 2394, probably due to a router failure. Similar temporary—but shorter—group disconnections are observed later on, around cycles 2450, 2550, and 2650, respectively.

Let us now examine the way the overlay is affected by those network changes. Figure 2.15 shows that the number of dead links is always kept at fairly small levels, especially when H is at least 1. As expected, the number of dead links peaks when there are massive node disconnections and gets back to normal quickly. However, it is not affected by the observed massive node reconnections, because these happen shortly after the respective disconnections, and the neighbors of the reconnected nodes are still alive.

Two observations regarding the effect of H can be made. First, higher values of H result in fewer dead links per view, validating the analysis in Section 2.5.2. Second, higher values of H trigger the *faster* elimination of dead links. The peaks caused by massive node disconnections are wider for low H values, and become sharper as H grows. In fact, these two observations are related to each other: in a persistently dynamic network, the converged average number of dead links depends on the rate at which the protocol disposes of them.

Figure 2.16 shows the evolution of the node degree deviation. It can be observed that for $H \geq 1$ the degree deviation under churn is very close to the corresponding converged values in a static network (see Figure 2.3). For $H = 0$ though, the higher number of

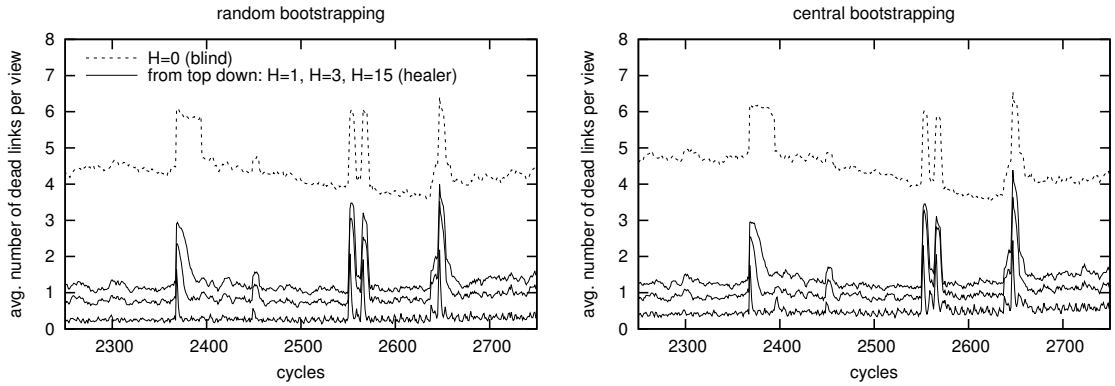


Figure 2.15: Average number of dead links per view, based on the Saroiu Gnutella traces. All experiments use random peer selection and $S = 0$.

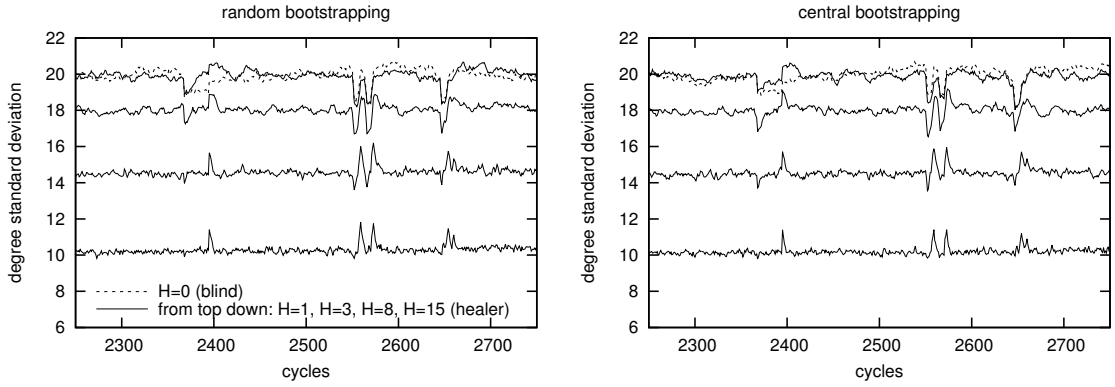


Figure 2.16: Evolution of standard deviation of node degree based on the Saroiu Gnutella traces. All experiments use random peer selection and $S = 0$.

pending dead links affects the degree distribution more. Note that both massive node disconnections *and* connections disturb the degree deviation, but in both cases a few cycles are sufficient to recover the original overlay properties.

To recap our analysis, we have shown that even with a pessimistic cycle length of 1 minute, all protocols for $H \geq 1$ perform very similarly to the case of a stable network, independently of the join method. Anomalies caused by massive node connections or disconnections are repaired quickly.

2.6 Wide-Area-Network Emulation

Distributed protocols often exhibit unexpected behavior when deployed in the real world that cannot always be captured by simulation. Typically, this is due to unexpected message loss, network and scheduling delays, as well as events taking place in unpredictable, arbitrary order. In order to validate the correctness of our simulation results, we implemented our gossip membership protocols and deployed them on a wide-area network.

We utilized the DAS-2 wide-area cluster as our testbed [72]. The DAS-2 cluster consists of 200 dual-processor nodes spread across 5 sites in the Netherlands. A total of 50 nodes were used for our emulations, 10 from each site. Each node was running a Java Virtual Machine emulating 200 peers, giving a total of 10,000 peers. Peers were running

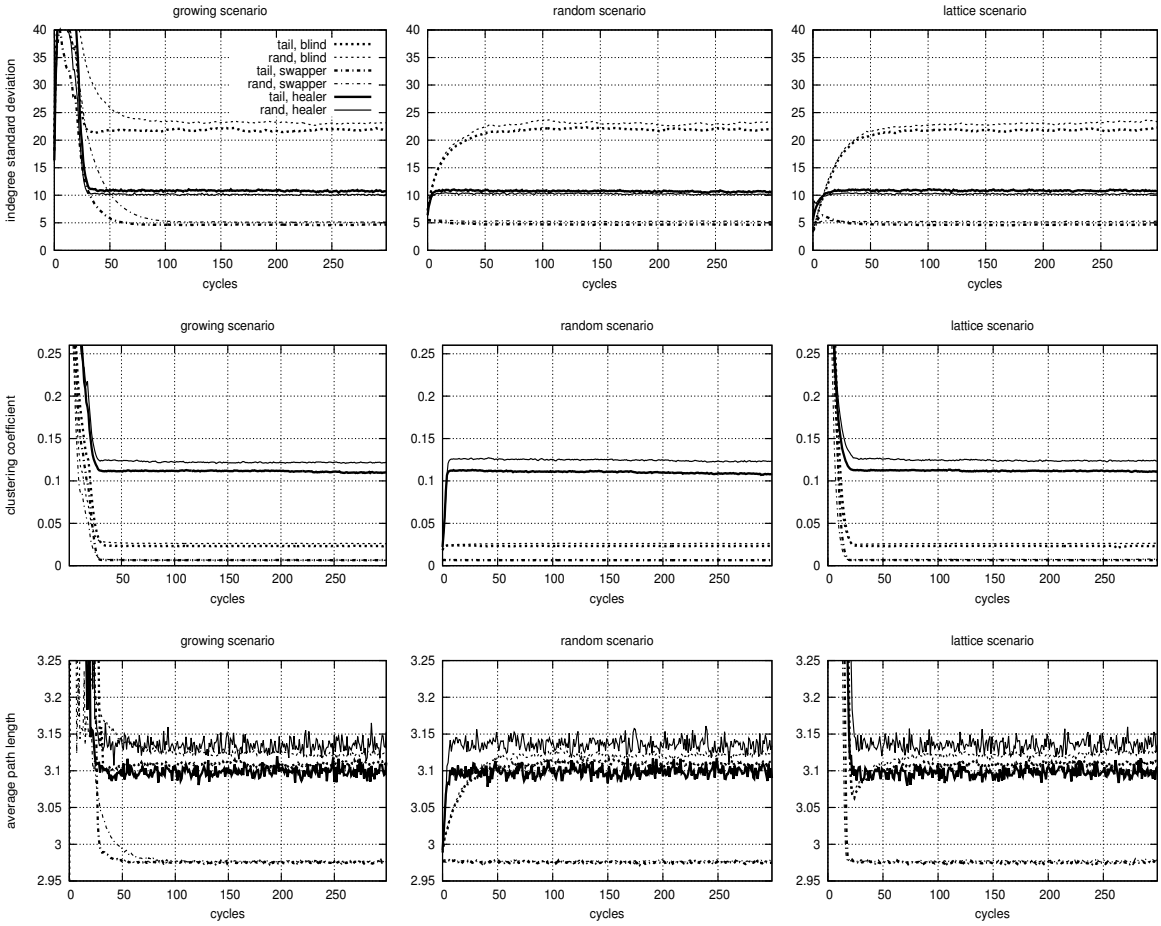


Figure 2.17: Evolution of indegree standard deviation, clustering coefficient, and average path length in all scenarios for *real-world* experiments.

in separate threads.

Although 200 peers were running on each physical machine, communication within a machine accounted for only 2% of the total communication. Local-area and wide-area traffic accounted for 18% and 80% of the total, respectively. Clearly, most messages are transferred through wide area connections. Note that the intra-cluster and inter-cluster round-trip delays on the DAS-2 are in the orders of 0.15 and 2.5 milliseconds, respectively. In all emulations, the cycle length was set to 5 seconds.

In order to validate our simulation results, we repeated the experiments presented in Figures 2.2 and 2.7 of Section 2.4, using our real implementation. A centralized coordinator was used to initialize the node views according to the bootstrapping scenarios presented in Section 2.4.1, namely *growing*, *lattice*, and *random*.

The first run of the emulations produced graphs practically indistinguishable from the corresponding simulation graphs. Acknowledging the low round-trip delay on the DAS-2, we ran the experiments again, this time inducing a 50 ms delay in each message delivery, accounting for a round-trip delay of 100 ms on top of the actual one. The results presented in this section are all based on these experiments.

Figure 2.17 shows the evolution of the indegree standard deviation, clustering coefficient, and average path length for all experiments, using the same scales as Figures 2.2 and 2.7 to facilitate comparison. The very close match between simulation-based and real-world experiments for all three nodes of the design space triangle allows us to claim

that our simulations represent a valid approximation of real-world behavior.

The small differences of the converged values with respect to the simulations are due to the induced round-trip delay. In a realistic environment, view exchanges are not atomic: they can be intercepted by other view exchanges. For instance, a node having initiated a view exchange and waiting for the corresponding reply, may in the meantime receive a view exchange request by a third node. However, the view updates performed by the active and passive thread of a node are not commutative. The results presented correspond to an implementation where we simply ignored this problem: all requests are served immediately regardless of the state of the serving node. This solution is extremely simple from a design point of view but may lead to corrupted views.

As an alternative, we devised and implemented three approaches to avoid corrupted views. In the first approach, a node's passive thread *drops incoming requests* while its active thread is waiting for a reply. In the second one, the node *queues*—instead of dropping—incoming requests until the awaited reply comes. As a third approach, a node's passive thread serves all incoming requests, but its active thread *drops a reply* if an incoming request intervened.

Apart from the added complexity that these solutions impose on our design, their benefit turned out to be difficult or impossible to notice. Moreover, undesirable situations may arise in the case of the first two: dropping or delaying a request from a third node may cause that node to drop or delay, in turn, requests it receives itself. Chains of dependencies are formed this way, which can render parts of the network inactive for some periods. Given the questionable advantage these approaches can offer, and considering the design overhead they impose, we will not consider them further. Based on our experiments, the best strategy is simply ignoring the problem, which further underlines the exceptional robustness and simplicity of gossip-based design.

2.7 Discussion

In this section we summarize and interpret the results presented so far. As stated in the introductory section, we were interested in determining the properties of various gossip membership protocols, in particular their randomness, load balancing and fault tolerance. In a sense, after we discussed in the last section why certain results were observed, we discuss here what the results imply.

2.7.1 Randomness

We have studied randomness from two points of view: local and global. Local randomness is based on the analogy between a pseudo random-number generator and the peer-sampling service as seen by a fixed node. We have seen that all protocols return a random sequence of peers at all nodes with a good approximation.

We have shown, however, that there are important correlations between the samples returned at different nodes, that is, the overlay graphs that the implementations are based upon are not random. Adopting a graph-theoretic approach, we have been able to identify important deviations from randomness that are different for the several instances of our framework.

In short, randomness is approached best by the view selection method SWAPPER ($H = 0, S = c/2 = 15$), irrespective of the peer selection method. In general, increasing H increases the clustering coefficient. The average path length is close to the one of a random

graph for all protocols we examined. Finally, with SWAPPER the degree distribution has a smaller variance than that of the random graph. This property can often be considered "better than random" (e.g., from the point of view of load balancing).

Clearly, the randomness required by a given application depends on the very nature of that application. For example, the upper bound of the speed of reaching all nodes via flooding a network depends exclusively on the diameter of the network, while other aspects such as degree distribution or clustering coefficient are irrelevant for this specific question. Likewise, if the sampling service is used by a node to draw samples to calculate a local statistical estimate of some global property, such as network size or the availability of some resources, what is needed is that the local samples are uniformly distributed. However, it is *not* required that the samples are independent at different nodes, that is, we do not need global randomness at all; the unstructured overlay can have any degree distribution, diameter, clustering, etc.

Load Balancing

We consider the service to provide good load balancing if the nodes evenly share the cost of maintaining the service and the cost induced by the application of the service. Both are related to the degree distribution: if many nodes point to a certain node, this node will receive more sampling-service related gossip messages and most applications will induce more overhead on this node, resulting in poor load balancing. Since the unstructured overlays that implement the sampling service are dynamic, it is also important to note that nodes with a high indegree become a bottleneck only if they keep having a high indegree for a long time. In other words, a node is in fact allowed to have a high indegree temporarily, for a short time period.

We have seen that the BLIND view selection is inferior to the other alternatives. The degree distribution has a high variance (that is, there are nodes that have a large indegree) and on top of that, the degree distribution is relatively static, compared to the alternatives.

Clearly, the best choice to achieve good load balancing is the SWAPPER view selection, which results in an even lower variance of indegree than in the uniform random graph. In general, the parameter S is strongly correlated with the variance of indegree: increasing S for a fixed H decreases the variance. The degree distribution is almost as static as in the case of HEALER, if $H = 0$. However, this is not a problem because the distribution has low variance.

Finally, HEALER also performs reasonably. Although the variance is somewhat higher than that of SWAPPER, it is still much lower than BLIND. Besides, the degree distribution is highly dynamic, which means that the somewhat higher variance of the degree distribution does not result in bottlenecks because the indegree of the nodes change quickly. In general, increasing H for a fixed value of S also decreases the variance.

Fault Tolerance

We have studied both catastrophic and realistic scenarios. In the first category, catastrophic failure and catastrophic churn were analyzed. In these scenarios, the most important parameter turned out to be H: it is always best to set H as high as possible. One exception is the experiment with the removal of 50% of the nodes, where SWAPPER performs slightly better. However, SWAPPER is slow in removing dead links, so if failure can be expected, it is highly advisable to set $H \geq 1$.

In the case of realistic scenarios, such as the realistic (artificial) churn rates, and the trace-based simulations, we have seen that the damaging effect is minimal, and (as long as $H \geq 1$) the performance of the protocols is very similar to the case when there is no failure.

2.8 Related Work

2.8.1 Gossip Membership Protocols

Most gossip protocols for implementing peer sampling are covered by our framework: we mentioned these in Section 2.2.4. One notable exception is [73] that we address here in some more detail. The protocol is as follows. In each cycle, all nodes pull the full partial views from F randomly selected peers. In addition, they record the addresses of the peers initiating incoming pull requests during the given cycle. The old view is then discarded and a new view is generated from scratch. In the most practical version, the new view is generated by first adding the addresses of the incoming requests and subsequently filling the rest of the view with random samples from the union of the previously pulled F views without replacement.

Notice that there are two features that are incompatible with our framework: the application of $F \geq 1$ (in our case $F = 1$) and the asymmetry between push and pull, with pull having a bigger emphasis. Only one entry—the initiator peer's own entry—is pushed. It is common to allow for $F \geq 1$ also in other proposals (e.g., [62]). In our framework, information exchange is symmetric, or fully asymmetric, without a finer tuning possibility.

To compare this protocol with our framework, we implemented it and ran simulations using our experimental scenarios. The view size and network size were the same as in all simulations, and F was 1, 2, or 3. The main conclusions are summarized below. The protocol class presented in [73] has some difficulty dealing with the scenarios when the initial network is not random (the growing and lattice initializations, see Section 2.4.1). For $F = 1$ we consistently observed partitioning in the lattice scenario (which was otherwise never observed in our framework). In the growing scenario—mostly for $F = 1$ but also for $F = 2$ and $F = 3$—the protocols occasionally get stuck in a local attractor where there is a star subgraph: a node with a very high indegree, and a large number of nodes with zero indegree and 1 as outdegree. Apart from these issues, if we consider self-healing, load balancing and convergence properties, the protocols roughly behave as if they were instances in our framework using pushpull, with $0 \leq H \leq 1$ and $S = 0$, with increasing F tending towards $H = 1$. Since we have concluded that the "interesting" protocols in our space have either a high H or a high S value, based on the empirical evidence accumulated so far there is no urgent need to extend our framework to allow for $F > 1$ or asymmetric information exchange. However, studying these design choices in more detail is an interesting topic for future research.

In the following we summarize a number of other fields that are relevant.

2.8.2 Complex Networks

The assumption of uniform randomness has only fairly recently become subject to discussion when considering large complex networks such as the hyperlinked structure of the WWW, or the complex topology of the Internet. Like social and biological networks,

the structures of the WWW and the Internet both follow the quite unbalanced power-law degree distribution, which deviates strongly from that of traditional random graphs. These new insights pose several interesting theoretical and practical problems [74]. Several dynamic complex networks have also been studied and models have been suggested for explaining phenomena related to what we have described here [75].

2.8.3 Unstructured Overlays

There are a number of protocols that are not gossip-based but that are potentially useful for implementing peer sampling. An example is the Scamp protocol [76]. While this protocol is reactive and so less dynamic, an explicit attempt is made towards the construction of a (static) random graph topology. Randomness has been evaluated in the context of information dissemination, and it appears that reliability properties come close to what one would see in random graphs. Some other protocols have also been proposed to achieve randomness [77, 78], although not having the specific requirements of the peer-sampling service in mind. Finally, random walks on arbitrary (hence, also unstructured) networks offer a powerful tool to obtain random samples, where even the sampling distribution can be adjusted [79]. These protocols, however, have a significantly higher overhead if many samples are required. This overhead and the convergence time also depend on the structure of the overlay network the random walk operates on.

2.8.4 Structured Overlays

In a sense, structured overlays have also been considered as a basic middleware service to applications [80]. However, a structured overlay [81–83] is by definition not dynamic. Hence utilizing it for implementing the peer-sampling service requires additional techniques such as random walks [79, 84]. Another example of this approach is a method assuming a tree overlay [85]. It is unclear whether a competitive implementation can be given considering also the cost of maintaining the respective overlay structure.

Another issue in common with our own work is that graph-theoretic approaches have been developed for further analysis [86]. Astrolabe [87] also needs to be mentioned as a hierarchical (and therefore structured) overlay, which, although applying (nonuniform) gossip to increase robustness and to achieve self-healing properties, does not even attempt to implement or apply a uniform peer-sampling service. It was designed to support hierarchical information aggregation and dissemination.

2.9 Concluding Remarks

Gossip protocols have recently generated a lot of interest in the research community. The overlays that result from these protocols are highly resilient to failures and high churn rates. The underlying paradigm is clearly appealing to build large-scale distributed applications

Our contribution is to factor out the abstraction implemented by the membership mechanism underlying gossip protocols: the peer-sampling service. The service provides every peer with (local) knowledge of the rest of system, which is key to have the system converge as a whole towards global properties using only local information.

We described a framework to implement a reliable and efficient peer-sampling service. The framework itself is based on gossiping. This framework is generic enough to be

instantiated with most current gossip membership protocols [5, 62, 63, 88]. We used this framework to empirically compare the range of protocols through simulations based on synthetic and realistic traces as well as implementations. We point out the very fact that these protocols ensure local randomness from each peer's point of view. We also observed that as far as the global properties are concerned, the average path length is close to the one in random graphs and that clustering properties are controlled by (and grow with) the parameter H. With respect to fault tolerance, we observe a high resilience to high churn rate and particularly good self-healing properties, again mostly controlled by the parameter H. In addition, these properties mostly remain independent of the bootstrapping approach chosen.

In general, when designing gossip membership protocols that aim at randomness, following a push-only or pull-only approach is not a good choice. Instead, only the combination results in desirable properties. Likewise, it makes sense to build in robustness by purposefully removing old links when exchanging views with a peer. This situation corresponds in our framework to a choice for $H > 0$.

Regarding other parameter settings, it is much more difficult to come to general conclusions. As it turns out, tradeoffs between, for example, load balancing and fault tolerance will need to be made. When focusing on swapping links with a selected peer, the price to pay is lower robustness against node failures and churn. On the other hand, making a protocol extremely robust will lead to skewed indegree distributions, affecting load balancing.

To conclude, we demonstrated in this extensive study that gossip membership protocols can be tuned to both support high churn rates and provide graph-theoretic properties (both local and global) close to those of random graphs so as to support a wide range of applications.

Chapter 3

Average Calculation

As computer networks increase in size, become more heterogeneous and span greater geographic distances, applications must be designed to cope with the very large scale, poor reliability, and often, with the extreme dynamism of the underlying network. *Aggregation* is a key functional building block for such applications: it refers to a set of functions that provide components of a distributed system access to global information including network size, average load, average uptime, location and description of hotspots, etc.

Local access to global information is often very useful, if not indispensable for building applications that are robust and adaptive. For example, in an industrial control application, some aggregate value reaching a threshold may trigger the execution of certain actions; a distributed storage system will want to know the total available free space; load balancing protocols may benefit from knowing the target average load so as to minimize the load they transfer.

In this chapter we elaborate on the aggregation protocol we introduced in Section 1.3. As mentioned there, the class of aggregate functions we can compute is very broad and includes many useful special cases such as counting, averages, sums, products and extremal values. The protocol is suitable for extremely large and highly dynamic systems due to its proactive structure—all nodes receive the aggregate value continuously, thus being able to track any changes in the system. The protocol is also extremely lightweight making it suitable for many distributed applications including peer-to-peer and grid computing systems. We demonstrate the efficiency and robustness of our gossip-based protocol both theoretically and experimentally under a variety of scenarios including node and communication failures.

3.1 Introduction

In this chapter, we focus on *aggregation* which is a useful building block in large, unreliable and dynamic systems [89] (see also Section 1.3). Aggregation is a common name for a set of functions that provide a summary of some global system property. In other words, they allow local access to global information in order to simplify the task of controlling, monitoring and optimization in distributed applications. Examples of aggregation functions include network size, total free storage, maximum load, average uptime, location and intensity of hotspots, etc. Furthermore, simple aggregation functions can be used as building blocks to support more complex protocols. For example, the knowledge of average load in a system can be exploited to implement near-optimal load-balancing schemes [61].

We distinguish *reactive* and *proactive* protocols for computing aggregation functions. Reactive protocols respond to specific queries issued by nodes in the network. The answers are returned directly to the issuer of the query while the rest of the nodes may or may not learn about the answer. Proactive protocols, on the other hand, continuously provide the value of some aggregate function to *all* nodes in the system in an *adaptive* fashion. By adaptive we mean that if the aggregate changes due to network dynamism or because of variations in the input values, the output of the aggregation protocol should track these changes reasonably quickly. Proactive protocols are often useful when aggregation is used as a building block for completely decentralized solutions to complex tasks. For example, in the load-balancing scheme cited above, the knowledge of the global average load is used by each node to decide if and when it should transfer load [61].

We introduce a robust and adaptive protocol for calculating aggregates in a proactive manner. We assume that each node maintains a local approximate of the aggregate value. The core of the protocol is a simple gossip-based communication scheme in which each node periodically selects some other random node to communicate with. During this communication the nodes update their local approximate values by performing some aggregation-specific and strictly local computation based on their previous approximate values. This local pairwise interaction is designed in such a way that all approximate values in the system will quickly converge to the desired aggregate value.

In addition to introducing our gossip-based protocol, the contributions are threefold. First, we present a full-fledged practical solution for proactive aggregation in dynamic environments, complete with mechanisms for *adaptivity, robustness* and *topology management*. Second, we show how our approach can be extended to compute complex aggregates such as variances and different means. Third, we present theoretical and experimental evidence supporting the efficiency of the protocol and illustrating its robustness with respect to node and link failures and message loss.

In Section 3.2 we define the system model. Section 3.3 describes the core idea of the protocol and presents theoretical and simulation results of its performance. In Section 3.4 we discuss the extensions necessary for practical applications. Section 3.5 introduces novel algorithms for computing statistical functions including several means, network size and variance. Sections 3.6 and 3.7 present analytical and experimental evidence on the high robustness of our protocol. Section 3.8 describes the prototype implementation of our protocol on PlanetLab and gives experimental results of its performance. Section 3.9 discusses related work. Finally, conclusions are drawn in Section 3.10.

3.2 System Model

We consider a network consisting of a large collection of *nodes* that are assigned unique identifiers and that communicate through message exchanges. The network is highly dynamic; new nodes may join at any time, and existing nodes may leave, either voluntarily or by *crashing*. Our approach does not require any mechanism specific to leaves: spontaneous crashes and voluntary leaves are treated uniformly. Thus, in the following, we limit our discussion to node crashes. Byzantine failures, with nodes behaving arbitrarily, are excluded from the present discussion (but see [90]).

We assume that nodes are connected through an existing routed network, such as the Internet, where every node can potentially communicate with every other node. To actually communicate, a node has to know the identifiers of a set of other nodes, called its *neighbors*. This neighborhood relation over the nodes defines the topology of an *overlay*

Algorithm 9 push-pull aggregation

```
1: loop
2:    wait(Δ)
3:    p ← selectPeer()
4:    sendPush(p,x)
5: procedure ONPUSH(m)
6:    sendPull(m.sender,x)
7:    x ← update(m.x, x)
8:
9: procedure ONPULL(m)
10:   x ← update(m.x, x)
```

network. Given the large scale and the dynamicity of our envisioned system, neighborhoods are typically limited to small subsets of the entire network. The set of neighbors of a node (thus the overlay network topology) can change dynamically. Communication incurs unpredictable delays and is subject to failures. Single messages may be lost, links between pairs of nodes may break. Occasional performance failures (e.g., delay in receiving or sending a message in time) can be seen as general communication failures, and are treated as such. Nodes have access to local clocks that can measure the passage of real time with reasonable accuracy, that is, with small short-term drift.

We focus on node and communication failures. Some other aspects of the model that are outside of the scope of the present analysis (such as clock drift and message delays) are discussed only informally in Section 3.4.

3.3 Gossip-based Aggregation

We assume that each node i in the network of N nodes holds a numeric value x_i. In a practical setting, this value can characterize any (possibly dynamic) aspect of the node or its environment (e.g., the load at the node, available storage space, temperature measured by a sensor network, etc.). The task of a proactive protocol is to continuously provide all nodes with an up-to-date estimate of an aggregate function, computed over the values held by the current set of nodes.

3.3.1 The Basic Aggregation Protocol

In Chapter 1 we have already presented push-pull averaging in Algorithm 4. For the sake of convenience, we repeat the algorithm here as Algorithm 9, with a slight generalization: instead of averaging the two values, the state update at the nodes is now expressed as an abstract method UPDATE. Method UPDATE computes a new local state based on the current local state and the remote state received during the information exchange. In most of this chapter, we limit the discussion to computing the average over the set of numbers distributed among the nodes, that is, method UPDATE(x, y) returns $(x + y)/2$. However, additional functions (most of them derived from the averaging protocol) are described in Section 3.5.

As of the peer sampling service, in Section 3.3.2 for theoretical reasons we will assume that SELECTPEER returns a true uniform random sample over the entire set of nodes. In Section 3.4.4 we revisit the peer sampling service from a practical point of view, by looking at realistic implementations based on non-uniform or dynamically changing overlay topologies.

Let us now consider the convergence of the protocol. It is easy to see that after one complete push-pull exchange, the sum of the two local estimates remains unchanged since

Algorithm 10 Avg

1: **for** $k = 1$ to N **do** ▷ vector $\mathbf{x}$ of length N is the input
2: $(i(k), j(k)) =$ getPair(k) ▷ perform elementary variance reduction step
3: $x_{i(k)} = x_{j(k)} = (x_{i(k)} + x_{j(k)})/2$
4: **return** $\mathbf{x}$

method UPDATE simply redistributes the initial sum equally among the two nodes; a property known as *mass conservation*. So, the operation does not change the global average but it decreases the variance over the set of all estimates in the system.

It is easy to see that the variance tends to zero in probability, that is, the value at each node will converge to the true global average in probability, as long as the network of nodes is not partitioned into disjoint clusters. To see this, one should consider the minimal value in the system. Clearly, if SELECTPEER returns uniform samples then in each cycle either the number of instances of the minimal value decreases or the global minimum increases with a probability of at least $1/N$ if there are different values from the minimal value (otherwise we are done because all values are equal). This is because if there is at least one different value, than any instance of the minimal value will get a neighbor with a different (thus larger) value with a probability of at least $1/N$.

The only non-trivial problem is to characterize the speed of the convergence of the expected variance. In the following, we will show that each cycle results in a reduction of the variance by a constant factor, which provides exponential convergence. We will assume that no failures occur and that the starting point of the protocol is synchronized. All of these assumptions will be relaxed later in the chapter.

3.3.2 Theoretical Analysis of Gossip-based Aggregation

We will treat the averaging protocol as an iterative variance reduction algorithm over a vector of numbers. To see how, consider that the distributed protocol in Algorithm 9 results in a series of push-pull exchanges between pairs of nodes. In fact, the behavior of the protocol is completely characterized by the series of node pairs that perform a push-pull exchange. This observation motivates the definition of Algorithm AVG (shown as Algorithm 10) that takes a vector $\mathbf{x}$ of length N as a parameter and produces a new vector $\mathbf{x}' = \text{AVG}(\mathbf{x})$ of the same length. The elements of the vector represent the local approximations at the nodes in the network of size N.

This centralized view of the protocol will let us develop the theoretical tools that will be used to characterize the original distributed protocol. Of course, in reality the push-pull exchanges in the network might overlap in time. For the sake of the theoretical discussion, we assume that the exchanges that involve a fixed node are non-overlapping in time, and thus these exchanges can be ordered. This defines a partial order that can always be extended to a total order. Any such extensions are equivalent from the point of view of convergence properties. Algorithm AVG represents the distributed execution via generating such a total order of communicating pairs via GETPAIR().

In this framework, we assume we are given an initial vector of numbers $\mathbf{x}(0) = (x_1(0) \ldots x_N(0))$. The elements of this vector correspond to the initial values at the nodes. The consecutive cycles of the protocol result in a series of vectors $\mathbf{x}(1), \mathbf{x}(2), \ldots$, where $\mathbf{x}(t+1) = \text{AVG}(\mathbf{x}(t))$. The behavior of our distributed gossip-based protocol can be reproduced by an appropriate implementation of GETPAIR. In addition, other implemen-

tations of GETPAIR are possible that do not necessarily map to any distributed protocol but are of theoretical interest. We will discuss some important special cases as part of our analysis.

Without loss of generality, to simplify our expressions, let us assume that the average of the values in the network is zero. Due to mass conservation, this will be true in all cycles:

$$\sum_{i=1}^{N} x_i(t) = 0, \quad t = 0, 1, \ldots \tag{3.1}$$

Under this assumption the variance in $\mathbf{x}$ is now given by

$$\sigma^2(t) = \frac{1}{N} \sum_{i=1}^{N} x_i^2(t). \tag{3.2}$$

Since the mean of the estimates remains constant (zero) due to mass conservation, from now on we can focus on $\sigma^2(t)$ as t tends to infinity. In particular, we want $\sigma^2(t)$ to quickly converge to zero because a small variance means that all nodes have a very accurate approximation.

Let us begin our analysis of the convergence of the variance with some basic observations. Let us have a look at the form of $\sigma^2(t+1)$ when expressed using the elements of $x(t)$. First, for illustration, consider $\sigma^2(t)'$ that is the variance of the vector after processing the first pair (i, j) returned by GETPAIR:

$$\begin{aligned} N\sigma^2(t)' &= x_1^2(t) + \cdots + \left(\frac{x_i + x_j}{2}\right)^2 + \cdots + \left(\frac{x_i + x_j}{2}\right)^2 + \cdots + x_N^2(t) \\ &= x_1^2(t) + \cdots + \frac{x_i^2}{2} + \cdots + \frac{x_j^2}{2} + \cdots + x_N^2(t) + x_i(t)x_j(t). \end{aligned} \tag{3.3}$$

Clearly, after completing the N cycles of algorithm AVG, we have

$$N\sigma^2(t+1) = \sum_{i=1}^{N} \left(\sum_{j=1}^{N} \alpha_{i,j} x_j(t) \right)^2 = \sum_{i=1}^{N} a_i x_i^2(t) + \sum_{i \neq j} b_{i,j} x_i(t) x_j(t), \tag{3.4}$$

where the parameters $\alpha_{i,j}$ (and thus a_i and $b_{i,j}$) are random variables that depend on the random decisions made by algorithm AVG.

We now discuss a few useful observations.

Proposition 3.3.1. *If algorithm* AVG *is symmetric to permutations (that is, for any permutation of the nodes π the series of pairs $(i(k), j(k))$ has the same probability as the series $(\pi(i(k)), \pi(j(k)), k = 1, \ldots, N)$ then for some constant a^* we have*

$$a^* = E(a_1) = \cdots = E(a_N) \tag{3.5}$$

(using the notations in Eq. (3.4)*). We will call a^* the* convergence factor. *We then have*

$$E(\sigma^2(t+1)) \leq a^* \sigma^2(t). \tag{3.6}$$

Proof. Eq. (3.5) follows directly from symmetry. Similarly, due to symmetry, it must be the case that for any $i \neq j$ and $m \neq n$: $E(b_{i,j}) = E(b_{m,n})$. Let b denote this common

constant. Then we have

$$\begin{aligned} E(\sigma^2(t+1)) &= \frac{1}{N}\sum_{i=1}^{N} E(a_i)x_i^2(t) + \frac{1}{N}\sum_{i\neq j} E(b_{i,j})x_i(t)x_j(t) \\ &= \frac{a^*}{N}\sum_{i=1}^{N} x_i^2(t) + \frac{b}{N}\sum_{i\neq j} x_i(t)x_j(t) \\ &= a^*\sigma^2(t) + \frac{b}{N}\sum_{i\neq j} x_i(t)x_j(t) \\ &= a^*\sigma^2(t) + \frac{b}{N}\sum_{i=1}^{N}\left(x_i(t)\sum_{j=1}^{N} x_j(t)\right) - \frac{b}{N}\sum_{i=1}^{N} x_i^2(t^2) \\ &= a^*\sigma^2(t) - b\sigma^2(t) \\ &\leq a^*\sigma^2(t), \end{aligned} \tag{3.7}$$

where we used Eq. (3.1) and the fact that $b \geq 0$, which follows from the fact that all the parameters $\alpha_{i,j}$ in (3.4) are non-negative. □

Proposition 3.3.2.

$$\sum_{i=1}^{N} a_i \geq \frac{N}{4} \tag{3.8}$$

for any fixed execution of AVG *with any implementation of method* GETPAIR *if* N *is even.*

Proof. Let us introduce the notations $a_i^{(k)}$ and $\alpha_{i,j}^{(k)}$ to represent the parameters that are analogous to a_i and $\alpha_{i,j}$ but in the state when only k cycles of AVG have been completed. Clearly, for all $i = 1, \ldots N$, $a_i^{(N)} = a_i$, $\alpha_{i,j}^{(N)} = \alpha_{i,j}$, $a_i^{(0)} = \alpha_{i,i}^{(0)} = 1$ and $\alpha_{i,j}^{(0)} = 0$ if $i \neq j$.

First, observe that when a pair (i, j) is picked by algorithm AVG in cycle k, then the contribution of node i to the difference $\sum_{i=1}^{N} a_i^{(k+1)} - \sum_{i=1}^{N} a_i^{(k)}$ is

$$2\sum_{j=1}^{N}\frac{1}{4}(\alpha_{i,j}^{(k)})^2 = \frac{1}{2}\sum_{j=1}^{N}(\alpha_{i,j}^{(k)})^2 \tag{3.9}$$

if the sets of non-zero α parameters of node i and j do not overlap, and strictly less if there is an overlap. Using this insight, let us observe that the maximal possible value of this contribution is $1/2$. This happens when node i is picked for the first time, because in this case the only non-zero α parameter is $\alpha_{i,i}^{(k)} = 1$. The second largest possible contribution is $1/4$. This can happen only when a node i is picked for the second time, and node i has exactly two non-zero α values (both having a value of $1/2$). To see this, consider that no α value can possibly be in the interval $(1/2, 1)$, and a node that is selected for the second time will have at least two non-zero α values.

From these observation we can see that the maximal overall difference $\sum_{i=1}^{N} a_i^{(N)} - \sum_{i=1}^{N} a_i^{(0)}$ is given by $N/2 + N/4$. The proposition directly follows from this, since $\sum_{i=1}^{N} a_i^{(0)} = N$. □

The assumption that N is even is extremely weak, given that we are interested in networks where N is very large, where this detail makes very little difference. Dealing with an odd N would not add much insight to the analysis, however, it would make the equations more complex, so we will not develop our results for that case.

Corollary 3.3.3. $a^* \geq 1/4$.

Remark 3.3.4. *The equality in Corollary 3.3.3 can be achieved, if* AVG *returns the $N/2$ pairs (let us assume that N is even) that form a perfect matching of the nodes as the first $N/2$ pair, followed by the $N/2$ pairs that form a second perfect matching that has no common pairs with the first perfect matching.*

Proposition 3.3.5. *If algorithm* AVG *is symmetric to permutation then*

$$E(\sigma^2(t+1)) = (1 - O(\frac{1}{N}))a^*\sigma^2(t). \tag{3.10}$$

Proof. Considering (3.7), where we have seen that $E(\sigma^2(t+1)) = (a^* - b)\sigma^2(t)$, we need to prove that $b/a^* = O(1/N)$. We first prove that $\sum_{i \neq j} b_{i,j} < N$. Let us consider the coefficients $\alpha_{i,j}$ and $b_{i,j}$ in Eq. (3.4). First of all, we know that on any node k we have $\sum_{i=1}^{N} \alpha_{k,i} = 1$; this follows from the mass conservation property of the algorithm. From this it follows that on any node k we have $\sum_{i \neq j} \alpha_{k,i}\alpha_{k,j} < 1$. Now we know that

$$\sum_{i \neq j} b_{i,j} = \sum_{i \neq j} \sum_{k=1}^{N} \alpha_{k,i}\alpha_{k,j} = \sum_{k=1}^{N} \sum_{i \neq j} \alpha_{k,i}\alpha_{k,j} < N. \tag{3.11}$$

Since $E(\sum_{i \neq j} b_{i,j}) = N(N-1)b$, it follows that $b < 1/(N-1)$. Based on Corollary 3.3.3 we have $b/a^* < 4/(N-1)$, which concludes the proof. □

This proposition indicates that the bound in (3.6) is tight in large networks. Having established the properties of the convergence factor—most importantly, the property that one needs to concentrate only on the quadratic terms in (3.4)—we can now give the expected value of the convergence factor for a number of interesting implementations of method GETPAIR, and ultimately, the convergence factor of Algorithm 9.

Pair Selection: Perfect Matching

As was discussed in Remark 3.3.4, the implementation of GETPAIR that is based on two perfect matchings is optimal, and results in a convergence factor of $a^* = 1/4$. We will call this implementation GETPAIR_PM where PM stands for perfect matching. This implementation cannot be mapped to an efficient distributed protocol directly because it requires global knowledge of the system. What makes it interesting is the fact that it is optimal,

Pair Selection: Random Choice

Moving towards more practical implementations of GETPAIR, our next example is GETPAIR_RAND which simply returns a random pair of different nodes independently for each call to GETPAIR, with all such pairs having an equal probability.

GETPAIR_RAND can easily be implemented as a distributed protocol, provided that SELECTPEER returns a uniform random sample of the set of nodes. When iterating AVG, the waiting time between two consecutive selections of a given node can be described by the exponential distribution. In a distributed implementation, a given node can approximate this behavior by waiting for a time interval randomly drawn from this distribution before initiating communication. However, as we shall see, GETPAIR_RAND is not a very efficient pair selector.

Theorem 3.3.6. *The limit of the convergence factor for* GETPAIR_RAND *is given by*

$$\lim_{N\to\infty} a^* = \frac{1}{e}. \tag{3.12}$$

Proof. First, we repeat the observation of the proof of Proposition 3.3.2 that when a pair (i, j) is picked by algorithm AVG in cycle k, then the contribution of node i to the difference $\sum_{i=1}^{N} a_i^{(k+1)} - \sum_{i=1}^{N} a_i^{(k)}$ is

$$2\sum_{j=1}^{N} \frac{1}{4}(\alpha_{i,j}^{(k)})^2 = \frac{1}{2}\sum_{j=1}^{N}(\alpha_{i,j}^{(k)})^2 \tag{3.13}$$

if the sets of non-zero α parameters of node i and j do not overlap.

In the case of GETPAIR_RAND, the α parameters will overlap only with a diminishing probability for a large N. This follows from the fact, that any node i will influence only a constant ($O(1)$) number of other nodes within one cycle on average, and conversely, any node is influenced only by a constant number of other nodes on average. So the probability that a node i gets a pair j with an overlapping set of α parameters has a probability $O(1/N)$.

Since originally the sum of the coefficients for the quadratic terms in the variance is $\sum_{j=1}^{N}(\alpha_{i,j}^{(k)})^2$, this means that node i reduces its actual contribution by a half every time it is picked. To be more precise, the remaining half contribution to the variance, $\frac{1}{2}\sum_{j=1}^{N}(\alpha_{i,j}^{(k)})^2$, will now be distributed among two nodes equally, with $\frac{1}{4}\sum_{j=1}^{N}(\alpha_{i,j}^{(k)})^2$ contributed by both node i and j.

However, since from a statistical point of view all the nodes have exactly the same future because GETPAIR_RAND makes decisions that are independent of the previous decisions, we can assume that the original contribution ($a_i^{(0)} = 1$) gets halved each time node i is picked. This will result in the same expected convergence factor. This convergence factor will then be given by the expectation

$$a^* = E(\frac{1}{2^\phi}) \tag{3.14}$$

where ϕ is a random variable that describes the number of times a node i was picked as a member of a pair. The distribution of ϕ can be approximated by the Poisson distribution with parameter 2 for a large N, that is

$$P(\phi = j) = \frac{2^j}{j!}e^{-2}. \tag{3.15}$$

Substituting this into the expression $E(2^{-\phi})$ we get

$$E(2^{-\phi}) = \sum_{j=0}^{\infty} 2^{-j}\frac{2^j}{j!}e^{-2} = e^{-2}\sum_{j=0}^{\infty}\frac{1}{j!} = e^{-2}e = e^{-1}. \tag{3.16}$$

□

Comparing the performance of GETPAIR_RAND and GETPAIR_PM we can see that convergence is significantly slower than in the optimal case (the factors are $e^{-1} \approx 1/2.71$ vs. $1/4$).

Pair Selection: a Distributed Solution

Building on the results we have so far, it is possible to analyze our original protocol described in Algorithm 9. If messages are delivered without delay then (assuming that the start of the cycle is not synchronized) nodes will wake up in a random order, and each of them will pair up with a random other node and complete one exchange.

In order to simulate this fully distributed version, the implementation of pair selection will return random pairs such that in each execution of AVG (that is, in each cycle), each node is guaranteed to be a member of at least one pair. This can be achieved by picking a random permutation of the nodes and pairing up each node in the permutation with another random node, thereby generating N pairs. We call this algorithm GETPAIR_DISTR. As we shall see, the performance of this protocol is superior to that of GETPAIR_RAND although of course does not match GETPAIR_PM that is optimal.

Theorem 3.3.7. *The limit of the convergence factor for* GETPAIR_DISTR *is given by*

$$\lim_{N\to\infty} a^* = \frac{1}{2\sqrt{e}}. \tag{3.17}$$

Proof. As in the proof of Theorem 3.3.6, we define ϕ to be the expected number of times a node participates in a pair. Random variable ϕ can be approximated as $\phi = 1+\phi'$ where ϕ' has the Poisson distribution with parameter 1, that is, for $j > 0$

$$P(\phi = j) = P(\phi' = j-1) = \frac{1}{(j-1)!}e^{-1}. \tag{3.18}$$

Again, similarly to the proof of Theorem 3.3.6, we calculate $E(2^{-\phi})$ and here we get

$$E(2^{-\phi}) = \sum_{j=1}^{\infty} 2^{-j}\frac{1}{(j-1)!}e^{-1} = \frac{1}{2e}\sum_{j=1}^{\infty}\frac{2^{-(j-1)}}{(j-1)!} = \frac{1}{2e}\sqrt{e} = \frac{1}{2\sqrt{e}}, \tag{3.19}$$

which is the desired formula.

The proof is not complete, however, because we cannot apply the reasoning of Theorem 3.3.6 in this case. The only difference is that here nodes might have different futures, since the pairs are not independent. In other words, different nodes might have a different expected number of times to be picked as a member of a pair in light of the number of times they have been selected before (except at the start of AVG, when no node has been selected yet).

In Theorem 3.3.6 we imagined the variance reduction process as a sort of stick breaking process in which a given initial unit contribution gets halved each time a node is selected (half of the stick is thrown away) and in addition, the remaining half stick is broken into two equal pieces and added to the contributions of the two members of the pair.

Let us divide the original unit stick into atoms such that when the stick is broken in two, half of the atoms are sent to the other node and half of them stay. Using 2^N atoms of initial size 2^{-N} suffices. In this view, an atom takes a random walk in the network, and half of its length is thrown away in each step. It makes one step each time the node it sits on is selected; in that case it stays where it is, or moves to the other node with a probability of $1/2$.

In this setting, we show that in the limit of large N the number of steps one atom makes (ϕ'') has the same distribution as ϕ, which is sufficient to complete the proof. We

do not present the complete proof here, because it is rather technical. The main idea is applying induction. First, we show that $P(\phi'' = 1) = 1/e$ in the limit of large N. In the inductive step we construct all the possible ways of making exactly $k+1$ steps assuming the possible walks (and their probability) of k steps are known. We then calculate the relative probability and show that $P(\phi'' = k+1)/P(\phi'' = k) = 1/k$ (in the limit of large N), which completes the proof, since through induction we get

$$P(\phi'' = k) = \frac{1}{(k-1)!}e^{-1}, \quad k > 0. \tag{3.20}$$

□

Comparing the performance of GETPAIR_DISTR to GETPAIR_RAND and GETPAIR_PM, we can see that convergence is slower than the optimal case but faster than random selection (the factors are $1/2\sqrt{e} \approx 1/3.3$, $e^{-1} \approx 1/2.71$ and $1/4$, respectively).

Empirical Results for Convergence of Aggregation

We ran AVG using GETPAIR_RAND and GETPAIR_DISTR for several network sizes and different initial distributions. For each parameter setting 50 independent experiments were performed.

Recall, that theory predicts that the average convergence factor is independent of the actual initial node values $\mathbf{x}(0)$. To test this, we initialized the nodes in two different ways. In the *uniform* scenario, each node is assigned an initial value uniformly drawn from the same interval. In the *peak* scenario, one randomly selected node is assigned a non-zero value and the rest of the nodes are initialized to zero.

Note that in the case of the peak scenario, methods that approximate the average based on a small random sample (that is, statistical sampling methods) are useless: one has to know all the values to calculate the average. Also, for a fixed variance, we have the largest difference between any two values. In this sense this scenario represents a worst case scenario. Last but not least, the peak initialization has important practical applications as well as we discuss in Section 3.5.

The results are shown if Figures 3.1 and 3.2. Figure 3.1 confirms our prediction that convergence is independent of network size and that the observed convergence factors match theory with very high accuracy. Note that smaller convergence factors result in faster convergence.

The only difference between the peak and the uniform scenario is that the variance of the convergence factor is higher for the peak scenario. Note that our theoretical analysis does not tackle the question of convergence factor variance. We can see however that the average convergence factor is well predicted and after a few cycles the variance is decreased considerably.

Figure 3.3 shows the difference between the maximal and the minimal estimates in the system for both the peak and uniform initialization scenarios. Note that although the expected variance $E(\sigma_i)$ decreases at the predicted rate, in the peak distribution scenario, the difference decreases faster. This effect is due to the highly skewed nature of the distribution of estimates in the peak scenario. In both cases, the difference between the maximal and the minimal estimate decreases exponentially and after as few as 20 cycles the initial difference is reduced by several orders of magnitude. This means that after a small number of cycles all nodes, including the outliers, will possess very accurate estimates of the global average.

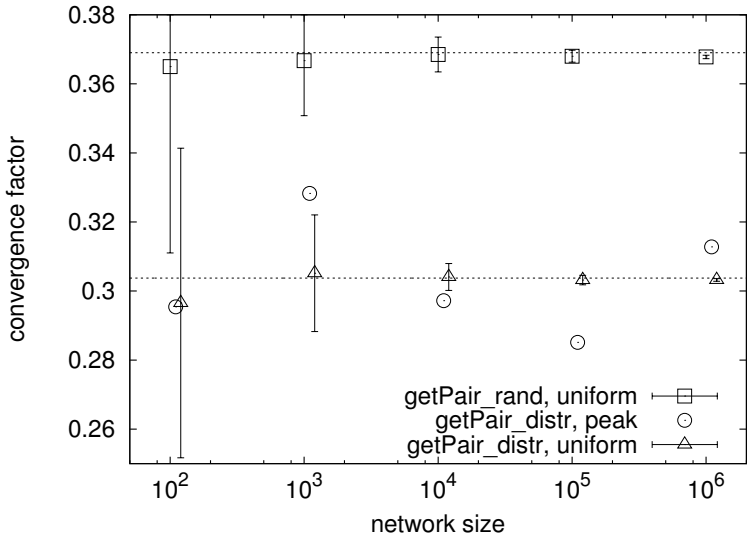


Figure 3.1: Convergence factor $\sigma^2(1)/\sigma^2(0)$ after one execution of AVG as a function of network size. For the peak distribution, error bars are omitted for clarity (but see Figure 3.2). Values are averages and standard deviations for 50 independent runs. Dotted lines correspond to the two theoretically predicted convergence factors: $e^{-1} \approx 0.368$ and $1/(2\sqrt{e}) \approx 0.303$.

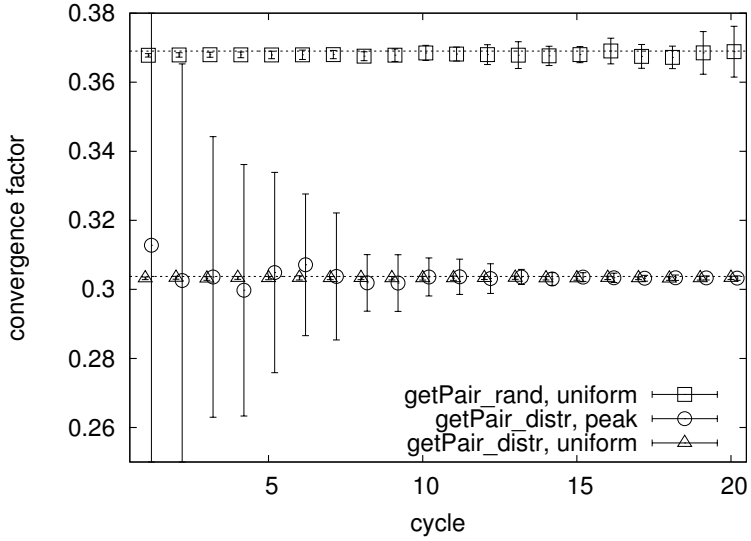


Figure 3.2: Convergence factor $\sigma^2(i)/\sigma^2(i-1)$ for network size $N = 10^6$ for different iterations of algorithm AVG. Values are averages and standard deviations for 50 independent runs. Dotted lines correspond to the two theoretically predicted convergence factors: $e^{-1} \approx 0.368$ and $1/(2\sqrt{e}) \approx 0.303$.

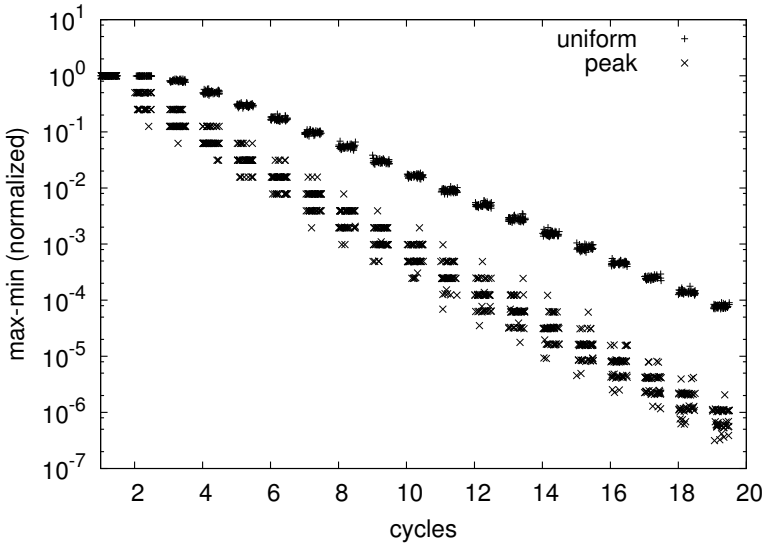


Figure 3.3: Normalized difference between the maximal and the minimal estimates as a function of cycles with network size $N = 10^6$. All 50 experiments are plotted as a single point for each cycle with a small horizontal random translation.

A Note on our Figures of Merit

Our approach for characterizing the quality of the approximations and convergence is based on the variance σ, and the convergence factor of the variance a^*, which describes the speed at which the expected value of σ decreases. To understand better what our results mean, it helps to compare it with other approaches to characterizing the quality of aggregation.

First of all, since we are dealing with a continuous process, there is no end result in a strict sense. Clearly, the figures of merit depend on how long we run the protocol. The variance $\sigma(i)$ characterizes the average *accuracy* of the approximates in the system in the given cycle i. In our approach, apart from averaging the accuracy over the system, we also average it over different runs, that is, we consider $E(\sigma(i))$. This means that an individual node in a specific run can have rather different accuracy. We have not considered the distribution of the accuracy (only the mean accuracy as described above), which depends on the initial distribution of the values. However, Figure 3.3 suggests that our approach is robust to the initial distribution.

Another frequently used measure is *completeness* [91]. This measure is defined under the assumption that the aggregate is calculated based on the knowledge of a subset of the values (ideally, based on the entire set, but due to errors this cannot always be achieved). It gives the percentage of the values that were taken into account. In our protocol this measure is difficult to adopt directly because at all times a local approximate can be thought of as a weighted average of the entire set of values. Ideally, all values should have equal weight in the approximations of the nodes (resulting in the global average value). To get a similar measure, one could characterize the distribution of weights as a function of time, to get a more fine-grained idea of the dynamics of the protocol.

3.4 A Practical Protocol for Gossip-based Aggregation

Building on the simple idea presented in the previous section, we now complete the details so as to obtain a full-fledged solution for gossip-based aggregation in practical settings.

3.4.1 Automatic Restarting

The generic protocol described so far is not adaptive, as the aggregation takes into account neither the dynamicity in the network nor the variability in values that are being aggregated. To provide up-to-date estimates, the protocol must be periodically *restarted*: at each node, the protocol is terminated and the current estimate is returned as the aggregation output; then, the current local values are used to re-initialize the estimates and aggregation starts again with these fresh initial values.

To implement termination, we adopt a very simple mechanism: each node executes the protocol for a predefined number of cycles, denoted as γ, depending on the required accuracy of the output and the convergence factor that can be achieved in the particular overlay topology adopted (see the convergence factor given in Section 3.3).

To implement restarting, we divide the protocol execution in consecutive *epochs* of length $\gamma\Delta$ (where Δ is the cycle length) and start a new instance of the protocol in each epoch. We also assume that messages are tagged with an epoch identifier that will be applied by the synchronization mechanism as described below.

3.4.2 Coping with Churn

In a realistic scenario, nodes continuously join and leave the network, a phenomenon commonly called churn. When a new node joins the network, it contacts a node that is already participating in the protocol. Here, we assume the existence of an out-of-band mechanism to discover such a node, and the problem of initializing the neighbor set of the new node is discussed in Section 3.4.4.

The contacted node provides the new node with the next epoch identifier and the time until the start of the next epoch. Joining nodes are not allowed to participate in the current epoch; this is necessary to make sure that each epoch converges to the average that existed *at the start* of the epoch. Continuously adding new nodes would make it impossible to achieve convergence.

As for node crashes, when a node initiates an exchange, it sets a timeout period to detect the possible failure of the other node. If the timeout expires before the message is received, the exchange step is skipped. The effect of these missing exchanges due to real (or presumed) failures on the final average will be discussed in Section 3.7. Note that self-healing (removing failed nodes from the system) is taken care of by the NEWSCAST protocol, which we propose as the implementation of method SELECTPEER (see Sections 3.4.4 and 3.7).

3.4.3 Synchronization

The protocol described so far is based on the assumption that cycles and epochs proceed in lock step at all nodes. In a large-scale distributed system, this assumption cannot be satisfied due to the unpredictability of message delays and the different drift rates of local clocks.

Given an epoch j, let T_j be the time interval from when the first node starts participating in epoch j to when the last node starts participating in the same epoch. In our protocol as it stands, the length of this interval would increase without bound given the different drift rates of local clocks and the fact that a new node joining the network obtains the next epoch identifier and start time from an existing node, incurring a message delay.

To avoid the above problem, we modify our protocol as follows. When a node participating in epoch i receives an exchange message tagged with epoch identifier j such that $j > i$, it stops participating in epoch i and instead starts participating in epoch j. This has the effect of propagating the larger epoch identifier (j) throughout the system in an epidemic broadcast fashion forcing all (slow) nodes to move up to the new epoch. In other words, the start of a new epoch acts as a synchronization point for the protocol execution forcing all nodes to follow the pace being set by the nodes that enter the new epoch first. Informally, knowing that push-pull epidemic broadcasts propagate super-exponentially (see Chapter 1) and assuming that each message arrives within the timeout used during all communications, we can obtain a logarithmic bound on T_j for each epoch j. More importantly, typically many nodes will start the new epoch independently with a very small difference in time, so this bound can be expected to be sufficiently small, which allows picking an epoch length such that it is significantly larger that T_j. A more detailed analysis of this mechanism would be interesting but is out of the scope of the present discussion. The effect of lost messages (i.e., those that time out) however, is discussed later.

3.4.4 Importance of Overlay Network Topology for Aggregation

The theoretical results described in Section 3.3 are based on the assumption that the underlying overlay is "sufficiently random". More formally, this means that the neighbor selected by a node when initiating communication is a uniform random sample among its peers. Yet, our aggregation scheme can be applied to generic connected topologies, by selecting neighbors from the set of neighbors in the given overlay network. This section examines the effect of the overlay topology on the performance of aggregation.

All of the topologies we examine (with the exception of NEWSCAST) are static—the neighbor set of each node is fixed. While static topologies are unrealistic in the presence of churn, we still consider them due to their theoretical importance and the fact that our protocol can in fact be applied in static networks as well, although they are not the primary focus of the present discussion.

Static Topologies

All topologies considered have a regular degree of 20 neighbors, with the exception of the complete network (where each node knows every other node) and the Barabási-Albert network (where the degree distribution is a power-law). For the random network, the neighbor set of each node is filled with a random sample of the peers.

The Watts-Strogatz and scale-free topologies represent two classes of realistic *small-world* topologies that are often used to model different natural and artificial phenomena [74, 92]. The Watts-Strogatz model [70] is obtained from a regular ring lattice. The ring lattice is built by connecting the nodes in a ring and adding links to their nearest neighbors until the desired node degree is reached. Starting with this ring lattice, each edge is then randomly *rewired* with probability β. Rewiring an edge at node n means

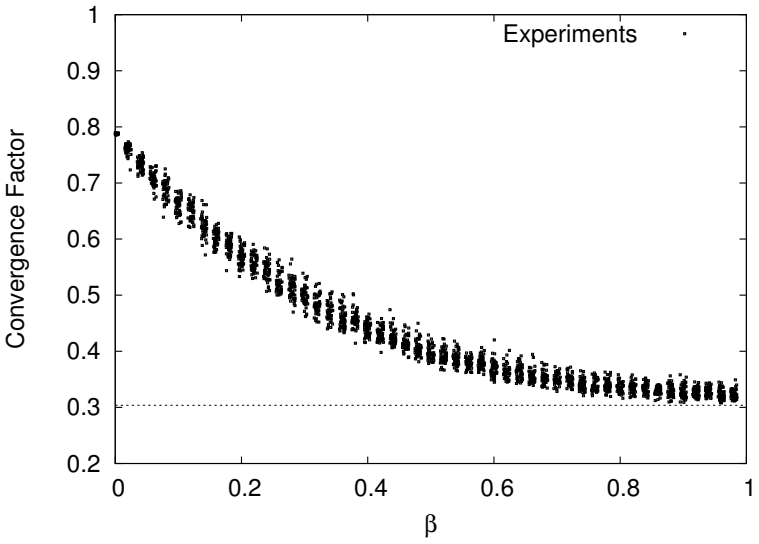


Figure 3.4: Convergence factor for Watts-Strogatz graphs as a function of parameter β. The dotted line corresponds to the theoretical convergence factor for peer selection through random choice: $1/(2\sqrt{e}) \approx 0.303$.

removing that edge and adding a new edge connecting n to another node picked at random. When $\beta = 0$, the ring lattice remains unchanged, while when $\beta = 1$, all edges are rewired, generating a random graph. For intermediate values of β, the structure of the graph lies between these two extreme cases: complete order and complete disorder.

Figure 3.4 focuses on the Watts-Strogatz model showing the convergence factor as a function of β ranging from 0 to 1. Although there is no sharp phase transition, we observe that increased randomness results in a lower convergence factor (faster convergence).

Scale-free topologies form the other class of realistic small world topologies. In particular, the Web graph, Internet autonomous systems, and P2P networks such as Gnutella [93] have been shown to be instances of scale-free topologies. We have tested our protocol over scale-free graphs generated using the preferential attachment method of Barabási and Albert [74]. The basic idea of preferential attachment is that we build the graph by adding new nodes one-by-one, wiring the new node to an existing node already in the network. This existing contact node is picked randomly with a probability proportional to its degree (number of neighbors).

Let us compare all the topologies described above. Figure 3.5 illustrates the performance of aggregation for different topologies by plotting the average convergence factor over a period of 20 cycles, for network sizes ranging from 10^2 to 10^6 nodes. Figure 3.6 provides additional details. Here, the network size is fixed at 10^5 nodes. Instead of displaying the average convergence factor, the curves illustrate the actual variance reduction (values are normalized so that the initial variance for all cases is 1) for the same set of topologies. We can conclude that performance is independent of network size for all topologies, while it is highly sensitive to the topology itself. Furthermore, the convergence factor is constant as a function of time (cycle), that is, the variance is decreasing exponentially, with non-random topologies being the only exceptions.

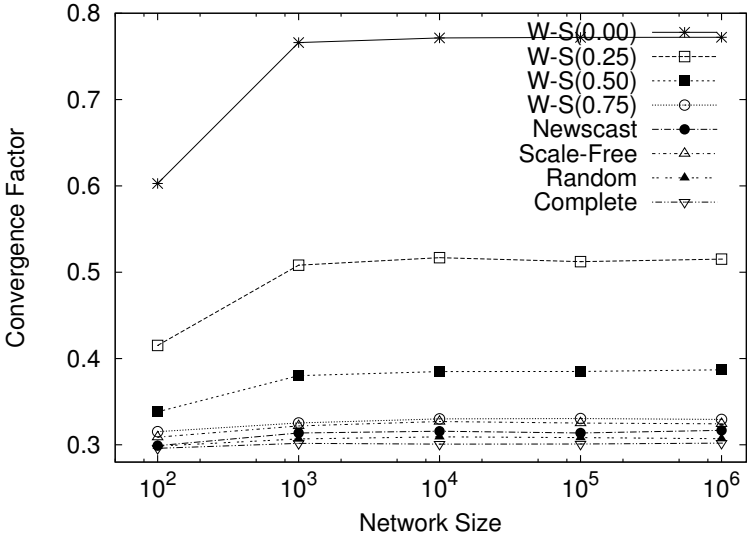


Figure 3.5: Average convergence factor computed over a period of 20 cycles in networks of varying size. Each curve corresponds to a different topology where W-S(β) stands for the Watts-Strogatz model with parameter β.

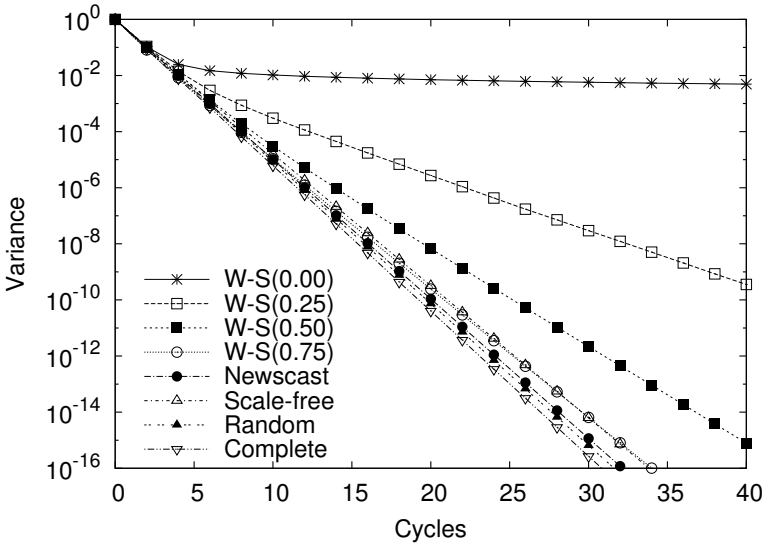


Figure 3.6: Variance reduction for a network of 10^5 nodes. Results are normalized so that all experiments result in unit variance initially. Each curve corresponds to a different topology where W-S(β) stands for the Watts-Strogatz model with parameter β.

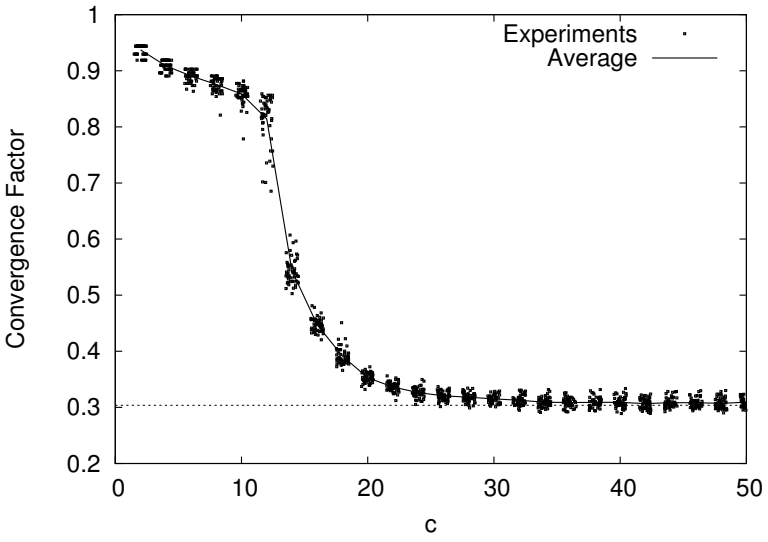


Figure 3.7: Convergence factor for NEWSCAST graphs as a function of parameter c. The dotted line corresponds to the theoretical convergence factor for peer selection through random choice: $1/(2\sqrt{e}) \approx 0.303$.

Dynamic Topologies

From the above results, it is clear that aggregation convergence benefits from increased randomness of the underlying overlay network topology. Furthermore, in dynamic systems, there must be mechanisms in place that preserve this property over time. To achieve this goal, we propose to use NEWSCAST, described in Section 2.2.4 (see Algorithm 8).

Recall that in NEWSCAST the overlay is generated by a continuous exchange of neighbor sets, where each element consists of a node identifier and a timestamp. These sets have a fixed size, which will be denoted by c. After an exchange, participating nodes update their neighbor sets by selecting the c node identifiers (from the union of the two sets) that have the freshest timestamps. Nodes belonging to the network continuously inject their identifiers in the network with the current timestamp, so old identifiers are gradually removed from the system and are replaced by newer information. This feature allows the protocol to "repair" the overlay topology by forgetting information about crashed neighbors, which by definition cannot inject their identifiers.

Figure 3.7 shows the performance of aggregation over a NEWSCAST network of 10^5 nodes, with c varying between 2 and 50. From these experimental results, choosing $c = 30$ is already sufficient to obtain fast convergence for aggregation. Furthermore, this same value for c is sufficient for very stable and robust connectivity (see Chapter 2). Figures 3.5 and 3.6 provide additional evidence that applying NEWSCAST with $c = 30$ already results in performance very similar to that of a random network.

3.4.5 Cost Analysis

Both the communication cost and time complexity of our scheme follow from properties of the aggregation protocol and are inversely related. The cycle length, Δ defines the time complexity of convergence. Choosing a short Δ will result in proportionally faster convergence but higher communication costs per unit time.

As we have seen earlier, if the overlay is sufficiently random then the number of exchanges for any fixed node in Δ time units can be described by the random variable $1 + \phi'$ where ϕ' has a Poisson distribution with parameter 1. Thus, on the average, there are two exchanges per node (one initiated by the node and the other one coming from another node), with a very low variance. Based on this distribution, parameter Δ must be selected to guarantee that, with very high probability, each node will be able to complete the expected number of exchanges before the next cycle starts. Failing to satisfy this requirement results in a violation of our theoretical assumptions.

Similarly, parameter γ (the epoch length) must be chosen appropriately, based on the desired accuracy of the estimate and the convergence factor a^* characterizing the overlay network. After γ cycles, we have $E(\sigma^2(\gamma))/\sigma^2(0) = a^{*\gamma}$. If ϵ is the desired accuracy of the final estimate, then $\gamma \geq \log_{a^*} \epsilon$. Note that a^* is independent of N, so the time complexity of reaching a given *average* precision is $O(1)$.

3.5 Aggregation Beyond Averaging

In this section we give several examples of gossip-based aggregation protocols to calculate different aggregates. With the exception of minimum and maximum calculation, they are all built on averaging. We also briefly discuss the question of dynamic queries.

3.5.1 Examples of Supported Aggregates

Minimum and maximum

To obtain the maximum or minimum value among the values maintained by all nodes, method UPDATE(a, b) of the generic scheme of Algorithm 9 must return $\max(a, b)$ or $\min(a, b)$, respectively. In this case, the global maximum or minimum value will be effectively broadcast like an epidemic. Well-known results about epidemic broadcasting [20] are applicable.

Generalized means

We formulate the general mean of a vector of elements $\mathbf{x} = (x_0, \ldots, x_N)$ as

$$f(\mathbf{x}) = g^{-1}\left(\frac{\sum_{i=0}^{N} g(x_i)}{N}\right) \tag{3.21}$$

where function f is the mean function and function g is an appropriately chosen local function to generate the mean. Well known examples include $g(x) = x$ which results in the average, $g(x) = x^n$ which defines the nth power mean (with $n = -1$ being the harmonic mean, $n = 2$ the quadratic mean, etc.) and $g(x) = \ln x$ resulting in the geometric mean (nth root of the product). To compute the above general mean, UPDATE(a, b) returns $g^{-1}[(g(a) + g(b))/2]$. After each exchange, the value of f remains unchanged but the variance over the set of values decreases so that the local estimates converge toward the general mean.

Variance and other moments

In order to compute the nth *raw* moment which is the average of the nth power of the original values, $\overline{x^n}$, we need to initialize the estimates with the nth power of the local value at each node and simply calculate the average. To calculate the nth central moment, given by $\overline{(x - \overline{x})^n}$, we can calculate all the raw moments in parallel up to the nth and combine them appropriately, or we can proceed in two sequential steps first calculating the average and then the appropriate central moment. For example, the variance, which is the 2nd central moment, can be approximated as $\overline{x^2} - \overline{x}^2$.

Counting

We base counting on the observation that if the initial distribution of local values is such that exactly one node has the value 1 and all the others have 0, then the global average is exactly $1/N$ and thus the network size, N, can be easily deduced from it. We will use this protocol, which we call COUNT, in our experiments.

Using a probabilistic approach, we suggest a simple and robust implementation of this scheme without any need for leader election: we allow multiple nodes to randomly start concurrent instances of the averaging protocol, as follows. Each concurrent instance is lead by a different node. Messages and data related to an instance are tagged with a unique identifier (e.g., the address of the leader). Each node maintains a map M associating a leader identifier with an average estimate. When nodes i and j maintaining the maps M_i and M_j perform an exchange, the new map M (to be installed at both nodes) is obtained by merging M_i and M_j in the following way:

$$\begin{aligned} M = &\{(l, x_i/2) \mid x_i = M_i(l) \in M_i \wedge l \notin D(M_j)\} \\ &\cup \{(l, x_j/2) \mid x_j = M_j(l) \in M_j \wedge l \notin D(M_i)\} \\ &\cup \{(l, (x_i + x_j)/2 \mid x_i = M_i(l) \wedge x_j = M_j(l)\}, \end{aligned} \tag{3.22}$$

where $D(M)$ corresponds to the domain (key set) of map M and x_i is the current estimate of node i. In other words, if the average estimate for a certain leader is known to only one node out of the two nodes that participate in an exchange, the other node is considered to have an estimate of 0.

Maps are initialized in the following way: if node l is a leader, the map is equal to $\{(l, 1)\}$, otherwise the map is empty. All nodes participate in the protocol described in the previous section. In other words, even nodes with an empty map perform random exchanges. Otherwise, an approach where only nodes with a non-empty set perform exchanges would be less effective in the initial phase while few nodes have non-empty maps.

Clearly, the number of concurrent protocols in execution must be bounded, to limit the communication costs involved. A simple mechanism that we adopt is the following. At the beginning of each epoch, each node may become leader of a run of the aggregation protocol with probability P_{lead}. At each epoch, we set $P_{\text{lead}} = C/\hat{N}$, where C is the desired number of concurrent runs and $\hat{N}$ is the estimate obtained in the previous epoch. If the systems size does not change dramatically within one epoch then this solution ensures that the number of concurrently running protocols will be approximately Poisson distributed with the parameter C.

Sums and products

Two concurrent aggregation protocols are run, one to estimate the size of the network, the other to estimate the average or the geometric mean, respectively. The size and the means together can be used to compute the sum or the product of the initial local values.

Rank statistics

Although the examples presented above are quite general, certain statistics appear to be difficult to calculate in this framework. Statistics that have a definition based on the index of values in a global ordering (often called *rank statistics*) fall into this category. While certain rank statistics like the minimum and maximum (see above) can be calculated easily, others, including the median, are more difficult. In our previous work we have proposed protocols for this purpose as well [13].

An example application: Naive Bayes

A central problem in data mining is classification. We assume that every node i has a *training data set* containing training samples. One sample consists of a *feature vector* $\mathbf{x} = (x_1, \ldots, x_r)$ and a *label* y. Let us assume that both the features x_i and y have a small discrete domain (indeed, for example, even binary features and binary labels are common). One wants to build a classification procedure that assigns labels to new observations (feature vectors) that are not labeled. This classification procedure might have a form of a decision tree, a regression formula, a description of a joint probability distribution, etc., [94]. Here, we will focus on a very simple, yet powerful, classification procedure called Naive Bayes.

The Naive Bayes procedure finds the maximum a posteriori (MAP) estimate

$$y_{MAP} = \arg\max_y p(y|\mathbf{x}) \tag{3.23}$$

with help of some empirical probabilities that are easy to find. Indeed, if we assume that attributes are conditionally independent with respect to the class attribute (it is a naive assumption therefore the name: Naive Bayes), the probabilities $p(y|\mathbf{x})$ can be expressed in terms of $p(x_i|y)$ and $p(y)$:

$$p(y|\mathbf{x}) = \frac{p(y)p(\mathbf{x}|y)}{p(\mathbf{x})} \approx \frac{p(y)\prod_i p(x_i|y)}{p(\mathbf{x})}. \tag{3.24}$$

The term $p(\mathbf{x})$ is not needed, since it is constant for a given feature vector, so it does not change the MAP estimate y_{MAP}.

Using our averaging protocol, we now need to calculate the average number of training samples (n), the average number of training samples that have y as label (n_y), and the average number of training samples that have feature x_i and label y ($n_{x_i,y}$). Now, each node can estimate

$$p(y) \approx \frac{n_y}{n}, \;\; p(x_i|y) \approx \frac{n_{x_i,y}}{n_y}. \tag{3.25}$$

Using these estimates, the MAP estimate y_{MAP} can be calculated. (Note, that n_y is redundant, but can nevertheless help in increasing stability.)

3.5.2 Dynamic Queries

Although here we target applications where the same query is calculated continuously and proactively in a highly dynamic large network, having a fixed query is not an inherent limitation of the approach. The aggregate value being calculated is defined by method UPDATE and the semantics of the state of the nodes (the parameters of method UPDATE). These components can be changed throughout the system at any time, using for example an extension of the restarting technique discussed in Section 3.4, where in a new epoch not only the start of the new epoch is being propagated through gossip but a new query as well.

Typically, our protocol will provide aggregation service for an application. The exact details of the implementation of dynamic queries (if necessary) will depend on the specific environment, taking into account efficiency and performance constraints and possible sources of new queries.

3.6 Theoretical Results for Benign Failures

3.6.1 Crashing Nodes

The result on convergence discussed in Section 3.3 is based on the assumption that the overlay network is static and that nodes do not crash. When in fact in a dynamic environment, there may be significant churn with nodes coming and going continuously. In this section we present results on the sensitivity of our protocols to dynamism of the environment.

Our failure model is the following. Before each cycle, a fixed proportion, say P_f, of the nodes crash (recall that we do not distinguish between nodes leaving the network voluntarily and those that crash). Given N nodes initially, $P_f N$ nodes are removed. We assume crashed nodes do not recover. Note that considering crashes only at the beginning of cycles corresponds to a worst-case scenario since the crashed nodes render their local values inaccessible when the variance among the local values is at its maximum. In other words, the more times a node communicates with other nodes, the better it approximates the correct global average (on average), so removing it at a latter stage does not disturb the end result as much as removing it at the beginning. Also recall that we are interested in the average at the beginning of the current epoch as opposed to the real-time average (see Section 3.4.1).

Let us begin with some simple observations. In our failure model the convergence factor will stay the same independently of P_f since the failure model is completely blind (there is no bias towards removing larger or smaller values), and the convergence factor does not depend on the network size N (as long as N is large). However, the average will now become a random variable that depends on P_f, since the mass conservation property no longer holds. Again, due to symmetry, it is trivial to see that the expectation of the average will not change (we still assume that it is zero). So, to characterizes the expected error of the approximation of the average, we consider the variance of the mean $\mathrm{Var}(\mu(t))$, where

$$\mu(t) = \frac{1}{N} \sum_{i=1}^{N(t)} x_i(t) \tag{3.26}$$

and $N(t) = (1 - P_f)^t N$. We will describe $\mathrm{Var}(\mu(t))$ as a function of P_f.

Proposition 3.6.1. *Let us assume that the convergence factor is a^* and algorithm* AVG *is symmetric to permutation. Then $\mu(t)$ has a variance*

$$Var(\mu(t)) \approx \frac{P_f}{(1-P_f)N}\sigma^2(0)\frac{1-\left(\frac{a^*}{1-P_f}\right)^t}{1-\frac{a^*}{1-P_f}}. \tag{3.27}$$

Proof. Let us take the decomposition $\mu(t+1) = \mu(t) + d_t$. Random variable d_t is independent of $\mu(t)$, because knowing only the average of a set does not contain information about the statistics of any strict subset (in this case, the subset that is removed) in the lack of additional prior information. So

$$\text{Var}(\mu(t+1)) = \text{Var}(\mu(t)) + \text{Var}(d_t), \tag{3.28}$$

which means that

$$\text{Var}(\mu(t)) = \sum_{j=0}^{t-1} \text{Var}(d_j). \tag{3.29}$$

This allows us to consider only $\text{Var}(d_t)$ as a function of failures. Note that $E(d_t) = 0$ since $E(\mu(t)) = E(\mu(t+1))$. Then, we have

$$\begin{aligned}\text{Var}(d_t) &= E((\mu(t)-\mu(t+1))^2) \approx \frac{P_f}{(1-P_f)N(t)}E(\sigma^2(t)) \\ &= \frac{P_f}{1-P_f}\sigma^2(0)\frac{a^{*t}}{N(t)} = \frac{P_f}{1-P_f}\sigma^2(0)\frac{a^{*t}}{N(1-P_f)^t}.\end{aligned} \tag{3.30}$$

which gives the desired formula when substituting (3.30) into (3.29). In the first equation we used the fact that $E(d_t) = 0$. The approximation then is the results of elementary calculations, in which we ignored the terms of the form ax_ix_j ($i \neq j$). Although here we do not formally quantify the error we make by ignoring these terms (instead, we perform an experimental validation) the considerations in the proofs of Propositions 3.3.1 and 3.3.5 strongly suggests that the error is not large. □

The results of simulations with $N = 10^5$ to validate this analysis are shown in Figure 3.8. For each value of P_f, the empirical data is based on 100 independent experiments whereas the prediction is obtained from (3.27) with $a^* = 1/(2\sqrt{e})$. The empirical data fits the prediction nicely. Note that the largest value of P_f examined was 0.3 which means that in each cycle almost one third of the nodes is removed. This already represents an extremely severe scenario. See also Section 3.7.1, where we present additional experimental analysis using NEWSCAST.

If $a^* > 1 - P_f$ then the variance is not bounded, it grows with the cycle index, otherwise it is bounded. Also note that increasing network size decreases the variance of the approximation $\mu(i)$. This is good news for scalability, as the larger the network, the more stable the approximation becomes.

3.6.2 Link Failures

In a realistic system, links fail in addition to nodes crashing. This represents another important source of error, although we note that from our point of view node crashes are more important because we model leaves as crashes, so in the presence of churn crash events dominate all other types of failure.

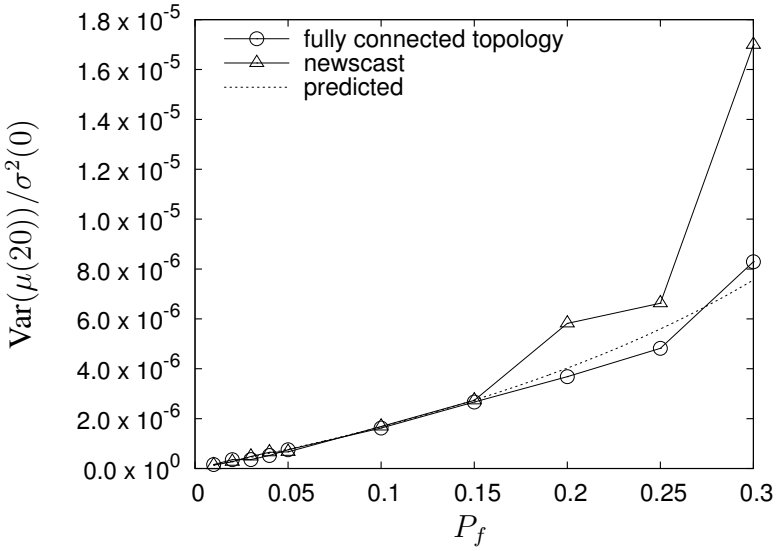


Figure 3.8: Effects of node crashes on the variance of the average estimates at cycle 20.

Let us adopt a failure model in which an exchange is performed only with probability $1 - P_d$, that is, each link between any pair of nodes is down with probability P_d. This model is adequate because we focus on short term link failures. For long term failures it is not sufficient to model failure as a probability, and long term failures can hardly be modeled as independent either. Besides, long term link failure in an *overlay* network means long term partitioning in the underlying physical network (because if the physical network was connected, normally the routing service could still function), and thus the overlay network is also partitioned. In such a partitioned topology our protocol will simply calculate an aggregate value local to each partitioned cluster.

In Section 3.3.2 it was proven that $a^* = 1/e$ (where a^* is the convergence factor) if we assume that during a cycle for each particular variance reduction step, each pair of nodes has an equal probability to perform that particular variance reduction step. For the protocol described in Algorithm 9 we have proven that $a^* = 1/(2\sqrt{e})$. For this protocol the uniform randomness assumption does not hold since the protocol guarantees that each node participates in at least one variance reduction step—the one initiated actively by the node. In the random model however, it is possible for example that a node does not participate in a given cycle at all.

Consider that a system model with $P_d > 0$ is very similar to a model in which $P_d = 0$ but which is "slower" (fewer pairwise exchanges are performed in a unit time interval). In the limit case when P_d is close to 1, the uniform randomness assumption described above (when $a^* = 1/e$) is fulfilled with high accuracy.

This motivates our conclusion that the performance can be bounded from below by the model where $P_d = 0$, and $a^* = 1/e$ instead of $1/(2\sqrt{e})$, and which is $1/(1-P_d)$ times slower than the original system in terms of wall clock time. That is, the upper bound on the convergence factor can be expressed as

$$a^*{}_d = (\frac{1}{e})^{1-P_d} = e^{P_d-1}. \tag{3.31}$$

Since the factor $1/e$ is not significantly worse than $1/(2\sqrt{e})$, we can conclude that practically only a proportional slowdown of the system is observed. In other words, link failures

do not result in any loss of approximation quality or increased unreliability.

3.6.3 Conclusions

We have examined two sources of random failures: node crashes and link failures. In the case of node crashes, the relationship was given between the proportion of failing nodes and the expected loss in accuracy of the average estimation. We have seen that the protocol can tolerate relatively large amounts of node crashes and still provide reasonable estimates. We have also shown that performance degrades gracefully with increasing link failure probability.

3.7 Simulation Results for Benign Failures

To complement the theoretical analysis, we have performed numerous experiments based on simulation. In all experiments, we used NEWSCAST as the underlying overlay network to implement function SELECTPEER in Algorithm 9. As a result, we need no unrealistic assumptions about the amount of information available at the nodes locally.

Furthermore, all our experiments were performed with the COUNT protocol since it is the aggregation example that is most sensitive to failures (both node crashes and message omissions) and thus represents a worst-case. During the first few cycles of an epoch when only a few nodes have a local estimate other than 0, their removal from the network due to failures can cause the final result of COUNT to diverge significantly from the actual network size.

All of experimental results were obtained through PEERSIM, a simulator developed by us and optimized for aggregation protocols [61, 66]. Unless stated otherwise, all simulations are performed on networks composed of 10^5 nodes. We do not present results for different network sizes since they display similar trends (as predicted by our theoretical results and confirmed by Figure 3.5).

The size of the neighbor sets maintained and exchanged by the NEWSCASTprotocol is set to 30. As discussed in Section 3.4.4, this value is large enough to result in convergence factors similar to those of random networks; furthermore, as our experiments confirm, the overlay network maintains this property also in the face of the node crash scenarios we examined. Unless explicitly stated, the size estimates and the convergence factor plotted in the figures are those obtained at the end of a single epoch of 30 cycles. In all figures, 50 individual experiments were performed for all parameter settings. When the result of each experiment is shown in a figure (e.g., as a dot) to illustrate the entire distribution, the x-coordinates are shifted by a small random value so as to separate results having similar y-coordinates.

3.7.1 Node Crashes

The crash of a node may have several possible effects. If the crashed node had a value smaller than the actual global average, the estimated average (which should be $1/N$) will increase and consequently the reported size of the network N will decrease. If the crashed node has a value larger than the average, the estimated average will decrease and consequently the reported size of the network N will increase.

The effects of a crash are potentially more damaging in the latter case. The larger the removed value, the larger the estimated size. At the beginning of an epoch, relatively

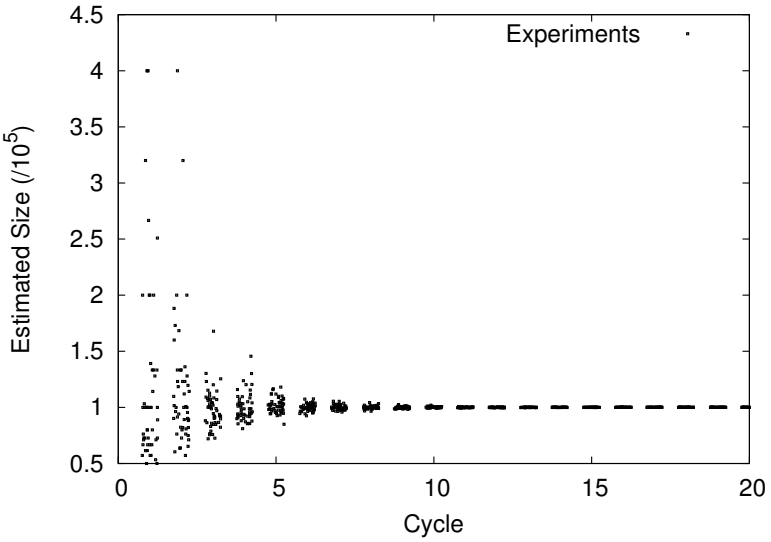


Figure 3.9: Network size estimation with protocol COUNT where 50% of the nodes crash suddenly. The x-axis represents the cycle of an epoch at which the "sudden death" occurs.

large values are present, obtained from the first exchanges originated by the initial value 1. These observations are confirmed by Figure 3.9, that shows the effect of the "sudden death" of 50% of the nodes in a network of 10^5 nodes at different cycles of an epoch. Note that in the first cycles, the effect of crashing may be very harsh: the estimate can even become infinite (not shown in the figure), if all nodes having a value different from 0 crash. However, around the tenth cycle the variance is already so small that the damaging effect of node crashes is practically negligible.

A more realistic scenario is a network subject to churn. Figure 3.10 illustrates the behavior of aggregation in such a network. Churn is modeled by removing a number of nodes from the network and substituting them with new nodes at each cycle. According to the protocol, the new nodes do not participate in the ongoing approximation epoch. However this scenario is not fully equivalent to a continuous node crashing scenario because these new nodes do participate in the NEWSCAST network and so they are contacted by participating nodes. These contacts are refused by the new nodes which results in an additional effect similar to link failure.

The size of the network is constant, while its composition is dynamic. The plotted dots correspond to the average estimate computed over all nodes that still participate in the protocol at the end of a single epoch (30 cycles), that is, that were originally part of the system at the start of the epoch. Note that although the average estimate is plotted over all nodes, in cycle 30 the estimates are practically identical as Figure 3.6 confirms. Also note that 2,500 nodes crashing in a cycle means that 75% of the nodes ($(30 \times 2500)/10^5$) are substituted during the epoch, leaving 25% of the nodes that make it until the end of the epoch.

The figure demonstrates that (even when a large number of nodes are substituted during an epoch) most of the estimates are included in a reasonable range. This is consistent with the theoretical result discussed in Section 3.6.1, although in this case we have an additional source of error: nodes are not only removed but replaced by new nodes. While the new nodes do not participate in the epoch, they result in an effect similar to link failure, as new nodes will refuse all connections that belong to the currently running epoch. How-

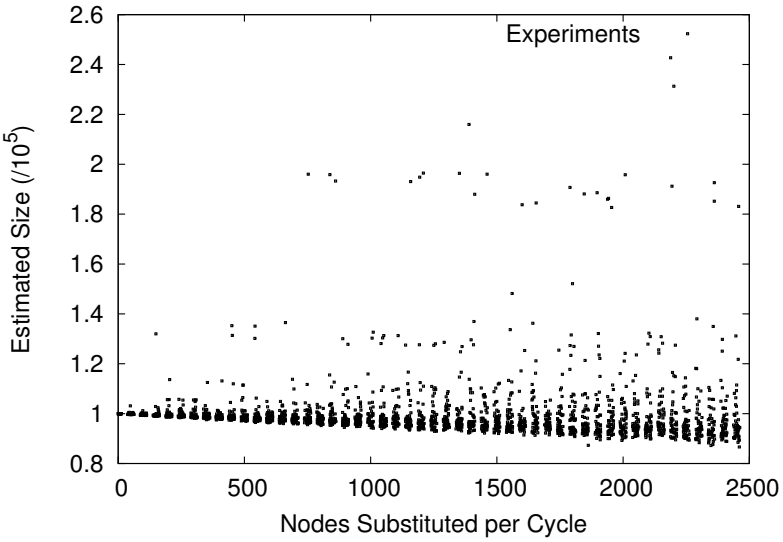


Figure 3.10: Network size estimation with protocol COUNT in a network of constant size subject to churn. The x-axis is the churn rate which corresponds to the number of nodes that crash at each cycle and are substituted by the same number of new nodes.

ever, the variance of the estimate continues to be described by the results in Section 3.6.1 because according to Sections 3.6.2 and 3.7.2, link failures do not change the estimate, only slows down convergence. Since an epoch lasts 30 cycles, this time is enough for convergence even beside the highest fluctuation rate. See also Figure 3.8 for the variance of the estimates plotted against the theoretical prediction.

The above experiment can be considered as a worst case analysis since the level of churn was much higher than it could be expected in a realistic scenario, considering that an epoch lasts for a relatively short time. We have repeated our experiments on the well-known Gnutella trace described in [58] to validate our results on a more realistic churn scenario as well. Figure 3.11 illustrates the simulation results. Only a short time window is shown (where the churn rate is particularly variable) to illustrate the accuracy of the approach better. We can observe that the approximation is accurate (with a one epoch delay), and the standard deviation is low as well. In this particular trace, during one epoch approximately 5% of the nodes are replaced. This is a relatively low rate and as we have seen earlier, the protocol can withstand much higher churn rates. Noted that the figure illustrates only the fluctuations in the network size as a result of churn and not the actual churn rate itself.

3.7.2 Link Failures and Message Omissions

Figure 3.12 shows the convergence factor of COUNT in the presence of link failures. As discussed earlier, in this case the only effect is a proportionally slower convergence. The theoretically predicted upper bound of the convergence factor (see (3.31)) indeed bounds the average convergence factor, and—as predicted—it is more accurate for higher values of P_d.

Apart from link failures that interrupt communication between two nodes in a symmetric way, it is also possible that single messages are lost. If the message sent to initiate an exchange is lost, the final effect is the same as with link failure: the entire exchange

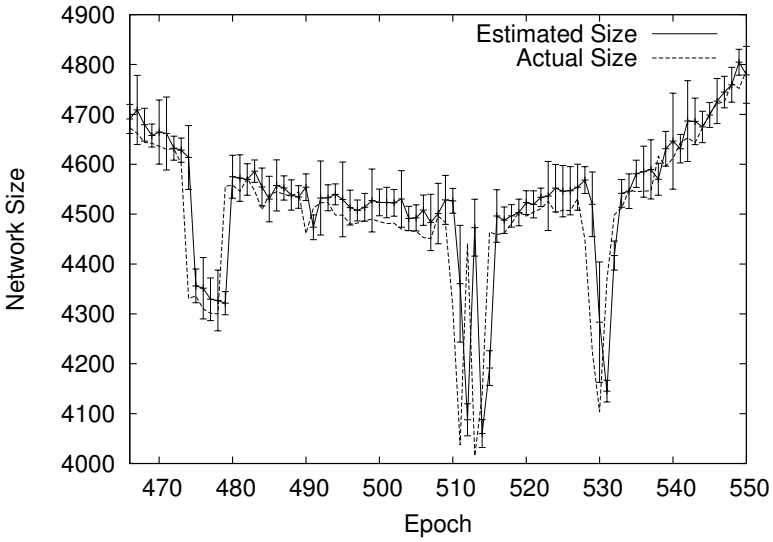


Figure 3.11: Network size estimation with protocol COUNT in the presence of churn according to a Gnutella trace [58]. 50 experiments were run to calculate statistics (mean and standard deviation), each epoch consisted of 30 cycles, each cycle lasted for 10 seconds.

is lost, and the convergence process is just slowed down. But if the message lost is the response to an initiated exchange, the global average may change (either increasing or decreasing, depending on the value contained in the message).

The effect of message omissions is illustrated in Figure 3.13. The given percentage of all messages (initiated or response) was dropped. For each experiment, both the maximum and the minimum estimates over the nodes in the network are shown, represented by the ends of the bars. As can be seen, when a small percentage of messages are lost, estimations of reasonable quality can be obtained. Unfortunately, when the number of messages lost is higher, the results provided by aggregation can be larger or smaller by several orders of magnitude. In this case, however, it is possible to improve the quality of estimations considerably by running multiple concurrent instances of the protocol, as explained in the next section.

3.7.3 Robustness via Multiple Instances of Aggregation

To reduce the impact of "unlucky" runs of the aggregation protocol that generate incorrect estimates due to failures, one possibility is to run multiple concurrent instances of the aggregation protocol. To test this solution, we have simulated a number t of concurrent instances of the COUNT protocol, with t varying from 1 to 50. At each node, the t estimates that are obtained at the end of each epoch are ordered. Subsequently, the $\lfloor t/3 \rfloor$ lowest estimates and the $\lfloor t/3 \rfloor$ highest estimates are discarded, and the reported estimate is given by the average of the remaining results.

Figure 3.14 shows the results obtained by applying this technique in a system where 1000 nodes per cycle are substituted with new nodes, while Figure 3.15 shows the results in a system where 20% of the messages are lost. Recall that even though in the node crashing scenario the number of nodes participating in the epoch decreases, the correct estimation is 10^5 as the protocol reports network size at the *beginning* of the epoch.

The results are quite encouraging; by maintaining and exchanging just 20 numerical

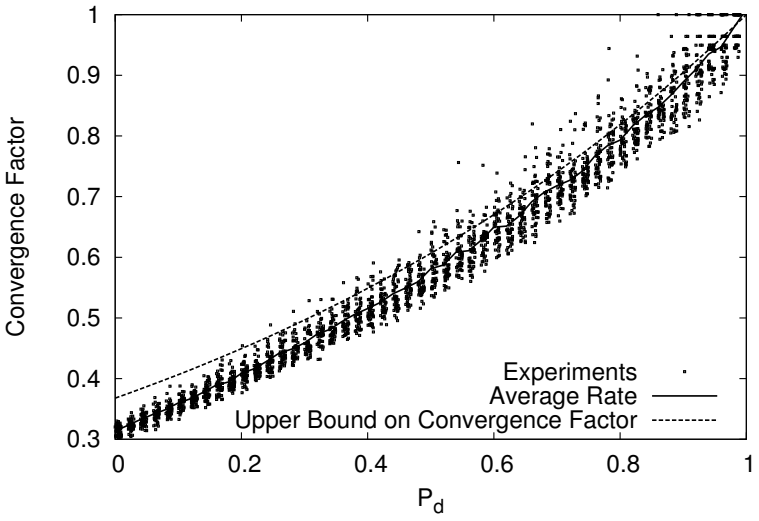


Figure 3.12: Convergence factor of protocol COUNT as a function of link failure probability.

values (resulting in messages of still only a few hundreds of bytes), the accuracy that may be obtained is very high, especially considering the hostility of the scenarios tested. It can also be observed that the estimate is very consistent over the nodes (the bars are short) in the crash scenario (as predicted by our theoretical results), and using multiple instances the variance of the estimate over the nodes decreases significantly even in the message omission scenario, so the estimate is sufficiently representative at every single node.

3.8 Experimental Results on PlanetLab

In order to validate our analytical and simulation results, we implemented the COUNT protocol and deployed it on PlanetLab [95]. PlanetLab is an open, globally distributed platform for developing, deploying and accessing planetary-scale network services. At the time of performing these experiments, more than 170 academic institutions and industrial research labs are members of the PlanetLab consortium, providing more than 400 nodes for experimentation.

A summary of the experimental results obtained on PlanetLab is illustrated in Figure 3.16. During the experiment, 300 machines belonging to the PlanetLab testbed were used. Each machine was running up to 20 virtual nodes, each participating as a distinct entity. In other words, the maximum size of our emulated network was 6000 virtual nodes, distributed over five continents. The size of the network was made to oscillate between 2500 and 6000 nodes during the experiment. Virtual nodes were removed and added using a central scheduler that randomly picked nodes from the network to produce the oscillation effect shown in the figure. The number of concurrent protocol instances was 20 (see Section 3.7.3), and parameter c of NEWSCAST was $c = 30$. The length of a cycle is 5 seconds, while the number of cycles in an epoch is 30 (that is, the length of an epoch is approximately 2.5 minutes). Several experiments were run, all of them starting at 02:00 Central European Time during workdays. All of them produced results similar to those shown in the figure. The communication mechanism of our implementation is

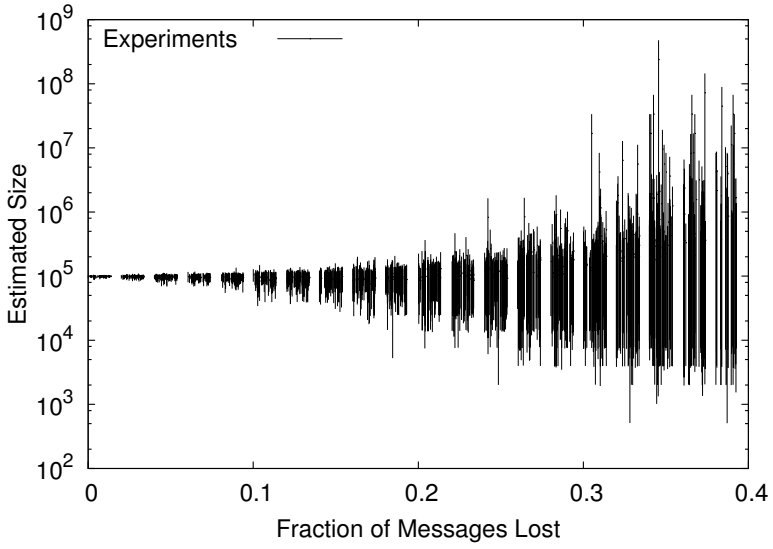


Figure 3.13: Network size estimation with protocol COUNT as a function of lost messages. The length of the bars illustrate the distance between the minimal and maximal estimated size over the set of nodes within a single experiment.

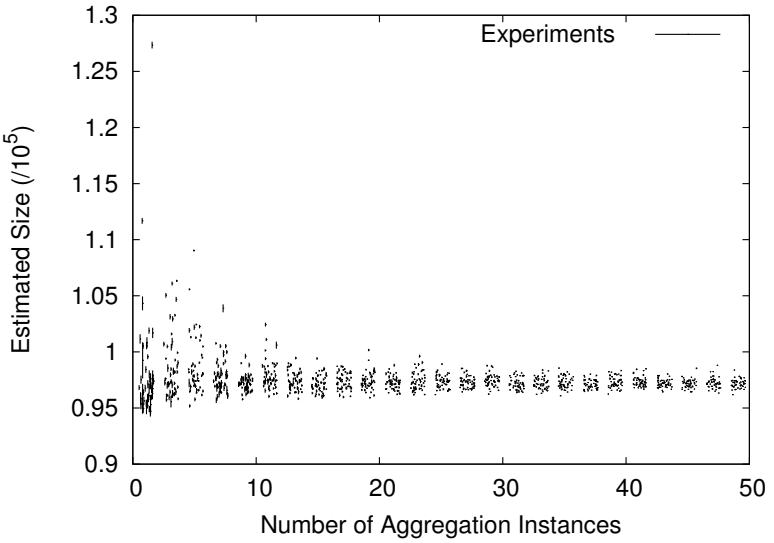


Figure 3.14: Network size estimation with multiple instances of protocol COUNT. 1000 nodes crash at the beginning of each cycle. The length of the bars correspond to the distance between the minimal and maximal estimates over the set of all nodes within a single experiment.

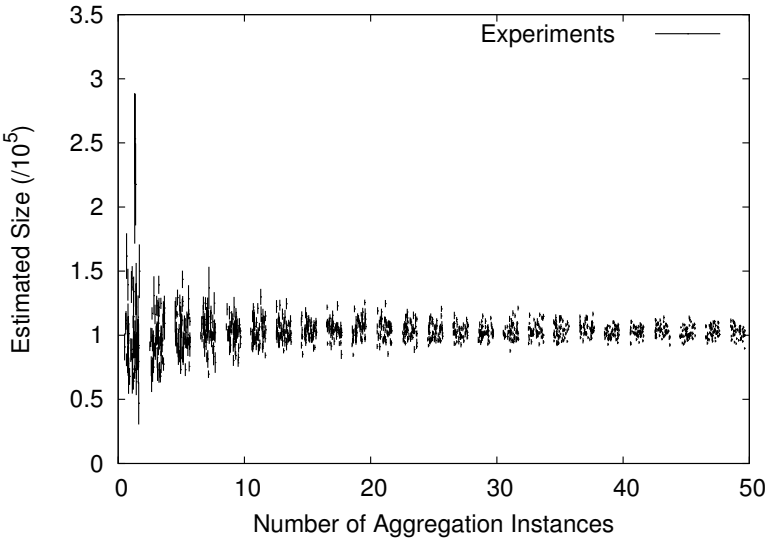


Figure 3.15: Network size estimation with protocol COUNT as a function of concurrent protocol instances. 20% of messages are lost. The length of the bars correspond to the distance between the minimal and maximal estimates over the set of all nodes within a single experiment.

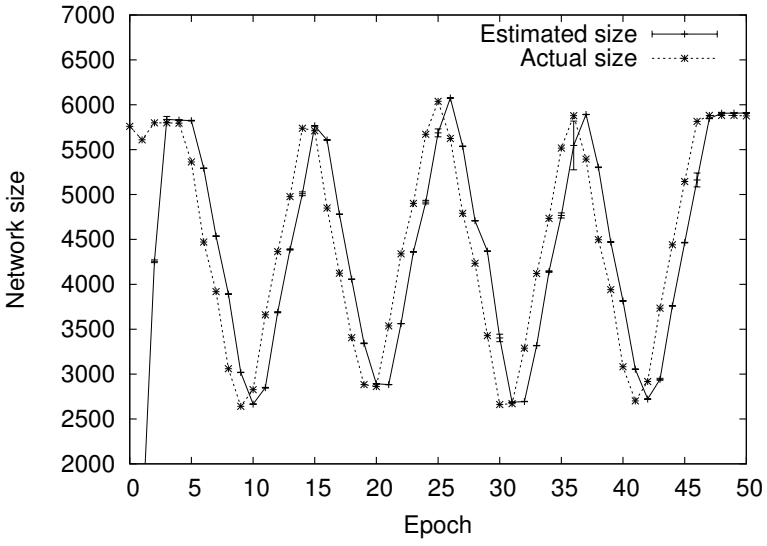


Figure 3.16: The estimated size (as provided by COUNT) and the actual size of a network oscillating between 2500 and 6000 nodes (approximately). Standard deviation of estimated size is displayed using vertical bars.

based on UDP. This choice is motivated by the fact that in a network based on NEWSCAST, interactions between nodes are short-lived, so establishing a TCP connection is relatively expensive. On the other hand, the protocol can tolerate message omissions. The observed message omission rate during our experiments varied between 3% and 8%.

The figure shows two curves, one representing the real size of the network at the beginning of a given epoch, and the other representing the estimated size, averaged over all nodes in the network. The (very small) standard deviation of the estimates over all nodes is also illustrated using vertical bars. These experiments further confirm the validity and practicality of our mechanisms.

3.9 Related Work

Since our work overlaps with a large number of fields, including gossip-based and epidemic protocols, load balancing, aggregation and network size estimation (in both overlay and wireless ad hoc networks), we restrict our discussion to the most relevant publications from each area.

Protocols based on epidemic and gossiping metaphors have found numerous practical applications. Examples include database replication [20] and failure detection [52]. A recently completed survey by Eugster et al. provides an excellent introduction to the area [59]. Note that our approach applies gossiping only as the communication model (periodic information exchange with random peers). Strictly speaking, nothing is "gossiped", the dynamics of the system is closer to a diffusion process. This is why, for example, theoretical results on epidemic spreading are not directly relevant here.

The load balancing protocol presented in [96] builds on the idea of generating a matching in the network topology and balancing load along the edges in the matching. Although the basic idea is similar, our work assumes a random overlay network (that we provide using NEWSCAST) and does not require the communications to take place in a matching in this network. Recall however that we have shown that the matching is the optimal case for our protocol; fortunately random pair selection has similar performance as well.

There are a number of general purpose systems for aggregation that offer a database abstraction (supporting queries about the state of the system) and that are based on structured (typically hierarchical) topologies. Perhaps the best-known example of this approach is Astrolabe [87], and more recently, SDIMS [97]. In these systems a hierarchical architecture is deployed which reduces the cost of finding the aggregates and enables the execution of complex database queries. However, maintenance of the hierarchical topology introduces additional overhead, which can be significant if the environment is very dynamic. Our gossip-based aggregation protocol is substantially different. Although the class of aggregates that it can compute is fairly general, and dynamic queries can also be implemented, it is not a general purpose system: it is extremely simple, lightweight, and targeted for unstructured, highly dynamic environments. Furthermore, our protocol is proactive: the updated results of aggregation are known to all nodes continuously.

The protocol presented in [91] suggests the so called Grid Box hierarchies to process queries in a structured fashion, which (compared to our protocol) involves increased message sizes and more complicated (so more vulnerable) execution which involves a logarithmic number of phases to calculate a single value. On the other hand, the overall approach is similar in the sense that all nodes are equivalent (run the same algorithm) and they all learn the end result.

Kempe et al. [34] propose an aggregation protocol similar to ours: it is based on gossiping and is tailored to work on random topologies. The main difference with the present work is that they consider push-only gossiping mechanisms, which results in a slightly more complicated (though still very simple) protocol. The complication comes from the fact that in a push-only approach some nodes attract more "weight" due to their more central position, so a normalization factor needs to be kept track of as well. Besides, other difficulties arise in practical settings if the directed graph used to push messages is not strongly connected. In our case the effective communication topology is undirected so we need only weak connectivity to allow the protocol to work. Furthermore, their discussion is limited to theoretical analysis, while we consider the practical details needed for a real implementation and evaluate their performance in unreliable and dynamic environments through simulations.

Related work targeted specifically to network size estimation should also be mentioned. A typical approach is to sample some property of the system which is random but depends on network size and so can be used to apply maximum likelihood estimation or a similar technique. This approach was followed in [98] in the context of multicasting. Another, probabilistic and localized technique is described in [99] where a logical ring is maintained and all nodes estimate network size locally based on the estimates of their neighbors. Unlike these approaches, our protocol provides the exact size in the absence of failures (assuming also that size is an integer which limits the necessary numeric precision) with very low cost and the approximation continues to be very accurate in highly unreliable and dynamic environments.

In principle, aggregation (even in the presence of malicious failures) could be achieved as follows: nodes run a protocol solving the agreement problem [100] (or the weaker approximate agreement problem [101, 102]) with their local values as the input. This suggests that the problems of aggregation and agreement are related. However, agreement protocols are designed for relatively small scale systems where the main problem is to deal with Byzantine failure. Agreement protocols are typically round based, requiring each node to communicate with every other node in a given interval of time (round). While the problem itself is similar, this approach is clearly not practical in the highly dynamic and extremely large scale settings we have in mind.

Finally, aggregation is an important problem in wireless and ad hoc networks as well. For instance, [103] represents a reactive approach where queries are propagated through the system and the answer propagates back to the source node (see the distinction between reactive and proactive approaches in the Introduction). The approach introduced in [104] is similar to ours. It is assumed that the network is a one-hop network (so all nodes can directly communicate with any other node), and a protocol is described that can manage the matching process that implements neighbor selection in this environment.

3.10 Conclusions

We have presented a full-fledged proactive aggregation protocol and have demonstrated several desirable properties including low cost, rapid convergence, robustness and adaptivity to network dynamics through theoretical an experimental analysis.

We proved that in the case of average calculation, the variance of the approximation of the average decreases exponentially fast, independently of network size. This result suggests both efficiency and scalability. We demonstrated that the method can be applied to calculate a number of aggregates beside the average. These include the maximum and

minimum, geometric and harmonic means, network size, sum and product. We proved theoretically that the protocol is not sensitive to node crashes, which confirms our approach of not introducing a leave protocol, but instead handling leaves as crashes. Link failures were also shown to only slightly slow down convergence.

The protocol was simulated on top of several different topologies, including random graphs, the complete graph, small-world networks like the Watts-Strogatz and Barabási-Albert topologies, and a dynamic adaptive unstructured network: NEWSCAST. It was demonstrated that the protocol is efficient on all of these topologies that have a small diameter.

We tested the robustness of the protocol in several failure scenarios. We have seen that very accurate estimates for the aggregate values can be obtained even if 75% of the nodes crash during the running of the protocol. Furthermore, it was confirmed empirically that the protocol is unaffected by link failures, which result only in a proportional slowdown but no loss in accuracy. Effects of single messages being lost are more severe but for reasonable levels of message loss, the protocol continues to provide highly-accurate aggregate values. Robustness to message loss can be greatly improved by the inexpensive and simple extension of running multiple instances of the protocol concurrently and calculating the final estimate based on the results of the concurrent instances. For node crashes and link failures, our experimental results are supported by theoretical analysis. Finally, the empirical analysis of the protocol was completed with emulations on PlanetLab that confirmed our theoretical and simulation results.

Chapter 4

Distributed Power Iteration

This chapter serves as our first example of a modular application of gossip components that work together to solve a relatively complex problem. As we will see, in this application the peer sampling service and gossip based averaging (described in Chapters 2 and 3, respectively) will both be used along with an asynchronous iteration algorithm.

The problem we tackle is determining the dominant eigenvector of matrices defined by weighted links in overlay networks. These eigenvectors play an important role in many peer-to-peer applications. Examples include trust management, importance ranking to support search, and virtual coordinate systems to facilitate managing network proximity. Robust and efficient asynchronous distributed algorithms are known only for the case when the dominant eigenvalue is exactly one. We present a fully distributed algorithm for a more general case: non-negative square matrices that have an arbitrary dominant eigenvalue.

The basic idea is that we apply a gossip-based aggregation protocol coupled with an asynchronous iteration algorithm, where the gossip component controls the iteration component. The norm of the resulting vector is an unknown finite constant by default; however, it can optionally be set to any desired constant using a third gossip control component. Through extensive simulation results on artificially generated overlay networks and real web traces we demonstrate the correctness, the performance and the fault tolerance of the protocol.

4.1 Introduction

The calculation of the dominant eigenvector of a matrix has long been a fundamental tool in almost all areas of science. In recent years, eigenvector calculation has found new and important applications in fully distributed environments such as peer-to-peer (P2P) overlay networks.

For example, the PageRank algorithm [105] calculates the *importance ranking* for hyperlinked web pages. The calculated ranks are given by the dominant eigenvector of a matrix that can be derived from the adjacency matrix of the graph defined by the hyperlinks. Fully distributed algorithms have already been proposed to implement PageRank [106–108]. As another example, *trust assignment* is a key problem in P2P networks. In [109] a method was proposed, that assigns a global trust value to each peer, through calculating the dominant eigenvector of the matrix containing local (pairwise) trust values. Finally, the eigenvectors that belong to the largest few absolute eigenvalues also play a role in esthetic low dimensional *graph layout* [110]. This application is relevant in vir-

tual coordinate assignment that allows to map the actual delays among all pairs of nodes onto the distance in the n-dimensional Euclidian space [111].

Motivated by these applications, and firmly believing that new ones will keep emerging, we identify fully distributed eigenvector calculation as an important P2P service that should be studied in its own right.

The environments we target impose special requirements. We assume, that there is a large number of nodes, the connections are volatile and unreliable and the eigenvector needs to be continuously updated and maintained in a decentralized way. Communication is implemented through message passing, where messages can be dropped or delayed. However, nodes have access to a local clock that measures the passage of real time with a reasonable accuracy. We do not assume that the local clocks at different nodes are synchronized.

In this model, we propose a protocol that involves three components. The first is an instantiation of the asynchronous iteration model described in [112]. This algorithm requires that the dominant eigenvalue is exactly one. We extend this protocol with a gossip-based control component that allows the iteration algorithm to converge even if the dominant eigenvalue is less than or greater than one. A third gossip component can be applied to explicitly control the exact value of the vector norm (which is an unspecified finite value without this third component).

These extensions make the asynchronous iteration robust to dynamic change and errors. Traditional methods are very sensitive to the dominant eigenvalue being exactly one: the slightest deviation results in misbehavior on the long run. Besides, the protocol is able to implement algorithms that assume a dominant eigenvalue different from one. A recent promising example is a ranking method using unnormalized web-graphs [113].

We demonstrate the correctness, the performance and the fault tolerance of the protocol through extensive simulation results on artificially generated overlay networks and real web traces.

4.2 Chaotic Asynchronous Power Iteration

Given a square matrix A, vector x is an *eigenvector* of A with *eigenvalue* λ, if $Ax = \lambda x$. Vector x is a *dominant* eigenvector if there are no other eigenvectors with an eigenvalue larger than $|\lambda|$ in absolute value. In this case λ is a *dominant eigenvalue* and $|\lambda|$ is the *spectral radius* of A.

We concentrate of the abstract problem of calculating the dominant eigenvector of a weighted neighborhood matrix of some large network, in a fully distributed way. By "fully distributed" we mean the worst case, when the elements of the vector are held by individual network nodes, one vector element per one node. The matrix A is defined by physical or overlay *links* between the network nodes, more precisely, the *weights* assigned to these links: let matrix element A_{ij} be the weight of the link from node j to node i. If there is no link from j to i then $A_{ij} = 0$.

In [112], Lubachevsky and Mitra present a chaotic asynchronous family of message passing algorithms to calculate the dominant eigenvector of a non-negative irreducible matrix, that has a spectral radius of one. Algorithm 11 shows an instantiation of this framework, that we will apply here.

In the algorithm, the values x_i represent the elements of the vector that converges to the dominant eigenvector. The values b_{ki} are buffered incoming weighted values from incoming neighbors in the graph. These values are not necessarily up-to-date, but, as

Algorithm 11 Asynchronous iteration executed at node i.

```
1: loop
2:   wait(Δ)
3:   for each j ∈ out-neighbors_i do
4:     send weight A_ji x_i to j
5:   b_i ← Σ_{k∈in-neighbors_i} b_ki
6:   x_i ← b_i
7: procedure ONWEIGHT(m)
8:   k ← m.sender
9:   b_ki ← m.x
```

shown in [112], the only assumption about message failure is that there is a finite upper bound on the age of these values. The age of value b_{ki} is defined by the time that elapsed since k sent the last update successfully received by i. This bound can be very large, so delays and message drop are tolerated extremely well. In addition, the values b_{ki} have to be initialized to be positive.

In dynamic scenarios, when nodes or network links are added or removed, the algorithm is still functional. Temporary node failures, churn, and link failures are all regarded as message failures, and are therefore covered by the assumption of the finite upper bound on update delay. Permanent changes can be dealt with as well: after the change the vector will start converging to the new eigenvector, provided simple measures are taken to make sure nodes remove dead links and take new ones into consideration.

4.3 Adding Normalization

Let λ_1 be a dominant eigenvalue of A. We can assume that $\lambda_1 \geq 0$ since A was assumed to be non-negative. The asynchronous method described above is known to work correctly if $\lambda_1 = 1$, but if $\lambda_1 > 1$ or $\lambda_1 < 1$, then the vector elements will grow indefinitely or tend to zero, respectively. This motivates us to propose a control component that continuously approximates the *average growth rate* of the vector elements, and normalizes each updated component with this value. Note that after we achieve convergence, the growth rate of every single vector element becomes λ_1. This suggests that approximating the global average using local, limited information is a viable plan.

We adopt the gossip protocol described in Chapter 3 to approximate the average growth rate. More precisely, we will use this algorithm to approximate the geometric mean of the local growth rates $b_i^{(m+1)}/x_i^{(m)}$ over all nodes i, where $b_i^{(m+1)}$ is the value calculated in line 5 in Algorithm 11 and $x_i^{(m)}$ is the value of x_i before executing line 6. The geometric mean is a more natural choice since we average multiplicative factors (growth rates).

The averaging protocol in Algorithm 9 is run by all nodes in parallel with the distributed power iteration. As of notation, let the gossip period of the averaging protocol be Δ_r and let r_i be the current approximation of the average at node i. As a result of the protocol, at all nodes these approximations quickly converge to the average of the initial values of the local approximations. The protocol relies on a peer sampling service, that returns a random node from the system. We use NEWSCAST to implement this service, a detailed description can be found in Section 2.2.4.

To calculate the geometric mean, each node i, when updating x_i, overwrites the local approximation of the growth rate by the *logarithm* of the locally observed growth rate of the vector element held by the node. That is, node i sets $r_i = \log(b_i^{(m+1)}/x_i^{(m)})$. The

approximation of the growth rate is therefore $e^{r_i(t)}$ at node i at time t. This value is used to normalize b_i, that is, we replace line 6 by $x_i = b_i/e^{r_i(t)}$ in the active thread of the iteration algorithm.

A cycle length $\Delta_r < \Delta$ is chosen so that a sufficiently accurate average is calculated, in spite of the continuous updates of r_i external to the averaging protocol. According to preliminary experiments, setting $\Delta_r = \Delta/5$ is already a safe choice on all the problems we examined. This is because, based on the results from Chapter 3, the approximation error decreases exponentially fast, besides, the growth rate is similar at all vector elements, as mentioned before.

4.4 Controlling the Vector Norm

The iteration component combined with gossip-based normalization is designed to achieve convergence, however, the norm of the converged vector is not known in advance. In some applications this might not be sufficient, since interpreting a single vector element becomes impossible, only relative values carry information. Besides, in scenarios when the matrix A constantly and frequently changes, the vector norm can grow without bounds or can tend to zero without explicitly controlling the vector norm. Finally, knowing a suitable vector norm makes it possible to implement some algorithms that require global knowledge. We will describe the random surfer operator of the PageRank algorithm as an example.

To address these issues, we apply a second gossip component for calculating the measure that we want to keep under control: for example, the maximum or the average of the absolute value of the vector elements. The calculation of these measures is accomplished by another instance of Algorithm 9, instantiated to calculate both the average and the maximum of the vector elements x_i (see Section 3.5.1).

Let the period of this component be Δ_n. The initial values in the normalization gossip component are updated at the same time when the growth rate gossip component updates its own initial values, as described in the previous section. It must be noted that in the case of norm calculation, $\Delta_n = \Delta/30$ appears to be necessary according to preliminary experiments, since, unlike growth rates, the vector elements themselves are not guaranteed to be similar, so we need to achieve very good convergence during a single period of the iteration algorithm.

Let us now present two examples for the application of the calculated average and maximum.

4.4.1 Keeping the Vector Norm Constant

Let us now assume that $n_i(t)$ is the approximation of either the maximum or the average of the vector at node i. To push this value towards one, we propose the following heuristic control factor to modify the normalization factor to introduce a bias towards the vector of which the average or maximum, respectively, is one. Intuitively, if $n_i(t)$ is too large, we decrease the local value a little more, and if it is too low, we increase a little more. More formally, we calculate a factor c as

$$c = e^{r_i(t)} \cdot \left(\frac{0.2}{1 + 1/n_i(t)} + 0.9 \right) \tag{4.1}$$

and subsequently replace line 6 with $x_i = b_i/c$. The factor c in (4.1) is a sigmoid function over the logarithm of $n_i(t)$, transformed to have range $[0.9, 1.1]$. This means that the growth rate approximation is never altered by more than 10% no matter how far the average is from one.

4.4.2 The Random Surfer Operator of PageRank

As a relatively more complex example of the possibilities this framework offers, we present the implementation of the random surfer operator used in the PageRank algorithm [105]. This operator will in turn allow us to implement PageRank as well.

The PageRank algorithm is concerned with the normalized adjacency matrix of a directed graph (e.g., the WWW link graph). Apart from this directed graph, the PageRank algorithm uses a "random surfer" operator R as well, defined as $R_{ij} = 1/N$, for all $i, j = 1, \ldots, N$, where N is the number of nodes. This corresponds to the definition of R as being a uniform random walk on the fully connected graph (hence the name "random surfer"). A very attractive feature of the random surfer operator is that it sends the *same* weight to every node. Hence there is no need to actually implement the all-to-all links, either in a centralized or in a distributed calculation. The net effect of R is to add a constant weight to each node at each propagation step. In other words, R times any vector gives a vector which is uniform, and whose value may be known if the average of the vector is known [105]. Hence, we can effectively replace matrix A with the PageRank operator $(1-\epsilon)A+\epsilon R$ where the second term involving R may be known as long as the average of $\mathbf{x}$ is known. Note that ϵ is a parameter of the PageRank algorithm, and defines the weight of the random surfer operator.

As described above, we can in fact obtain an approximation of the vector average. Then we can implement the PageRank R operator—a global operator—using purely local operations: node i now has the update rule

$$x_i = (1-\epsilon)\frac{b_i}{e^{r_i(t)}} + \epsilon n_i(t), \tag{4.2}$$

where $n_i(t)$ is the locally known converged approximation of the average at time t. Finally, note that controlling the average of the vector and applying the random surfer operator can be done simultaneously as well, using the update rule

$$x_i = (1-\epsilon)\frac{b_i}{c} + \epsilon n_i(t), \tag{4.3}$$

where c is defined as in (4.1).

4.5 Experimental Results

We performed extensive event-based simulation experiments using the PEERSIM simulator [66]. The goal of the experiments was to demonstrate that our method is both efficient and robust to failure.

4.5.1 Notes on the Implementation

In the case of one of the components—the gossip-based protocol that continuously calculates the *average* of the current vector approximation, described in Section 4.4—we applied two modifications to increase robustness.

First, instead of Algorithm 9, we applied push averaging presented in [34] and in Section 1.3.1. This variant is very similar; the main difference is that it is slightly modified so that it can apply the "push only" communication model, while the original version is based on the "push-pull" model. In the push model, the nodes only send messages, but need not answer them. In the push-pull model all messages must be answered immediately. We apply the push variant because it is more robust to message delays: while in the push-pull version the state of the nodes are inconsistent for a short time (between the sending and reception of the answer), this problem does not exist in the push version.

The second modification of this component is that we apply the epoch-based restarting technique described in Chapter 3. This is necessary to prevent nodes from mixing converged values with freshly initialized ones, since otherwise we could never achieve convergence, since the local values are constantly re-initialized by the iteration component.

4.5.2 Artificially Generated Matrices

For evaluating the protocol we applied a set of artificially generated matrices with controlled properties. To model real applications mentioned in the Introduction, all matrices are sparse and are derived from the adjacency matrix of a link graph. First let us define the graphs that were used to define the matrices.

All graphs have 5000 nodes. The baseline case is a directed random graph, according to the k-out model. In this model, k random out-links are added to each node. We generated an instance of this model with $k = 8$.

The second graph is a scale free graph generated by the Barabási-Albert model [114]. Most importantly, the degree distribution of this graph follows a power law, that is extremely unbalanced with many low degree nodes and a few high degree nodes, and that is known to describe many interesting emergent networks such as the WWW or social relationships [114]. The parameter of the model was set to two. In this case the Barabási-Albert model defines an undirected graph by starting with two disconnected nodes, and subsequently adding nodes one by one, linking each new node to two existing nodes. These two nodes are selected with a probability proportional to their degree (preferential attachment). The average degree in the graph is thus four.

The third graph was generated starting with an undirected ring, and adding two random out-links from all nodes (note that this procedure follows a modified version of the Watts-Strogatz model [70]). The motivation behind using this graph is that, as we will see, its adjacency matrix has a small eigenvalue gap, which results in a slow convergence of the power iteration. This graph was chosen to test whether our method is sensitive to a small eigenvalue gap.

The matrices were derived from the adjacency matrices of these graphs. We note for completeness that the specific instances of the directed graphs we used (the random k-out and the small gap graphs) were all strongly connected. Since our convention is that the element A_{ij} describes the weight for network link (j, i)—so that matrix vector multiplication can be defined by sending messages along the outgoing (and not incoming) links—the adjacency matrices were first transposed. The first set of matrices consists of the transposed adjacency matrices. The second set contains the column normalized versions of the matrices in the first set. The normalized versions are such that the weights of the outgoing links sum up to one for all the nodes, therefore these matrices describe random walks on the graphs.

	random k-out		scale free		small gap	
	normalized	unnorm.	normalized	unnorm.	normalized	unnorm.
λ_1	1.0000	8.0000	1.0000	1.3981	1.0000	4.1938
$\lvert\lambda_2\rvert$	0.3573	2.8345	0.8373	1.1737	0.9754	3.9976

Table 4.1: The first and second largest magnitude eigenvalues. Note that the largest magnitude eigenvalue is guaranteed to be real and positive.

Table 4.1 shows the first two largest magnitude eigenvalues for all the problem instances. Note that the eigenvalue gap (the difference between the first and second largest eigenvalues) determines the convergence speed of the power iteration [115], and thus it is a good indicator of the speed of our method as well. For a small gap, convergence is slow. With a zero gap, the power iteration does not converge at all. Since all matrix elements are real and non-negative, the largest eigenvalue is real and non-negative as well.

4.5.3 Results

Each experiment was carried out as follows. First, each node i was initialized to have $x_i = 1$ and $b_i = 1$. As we explained previously, the starting time of individual nodes is irrelevant from the point of view of convergence results, as long as all nodes start eventually. In the simulations we started each node at a random time within the first Δ time units counted from the first snapshot time t_0.

Two versions of the method were run for each problem. In the first, we do not apply the vector normalization gossip component described in Section 4.4. In this case we expect the vector norm to converge to a previously unknown value, given that we do not change the underlying matrices in these experiments. In the second version we do apply vector normalization. In particular, we apply the *maximum* of the vector for this purpose, and therefore we expect the maximum to converge to one.

The evaluation metrics were as follows. We first computed the correct dominant eigenvector ($\mathbf{x}^*$) using a centralized algorithm. Following general practice in matrix computations, we measured the angle of the actual approximation and the correct vector to characterize convergence. That is, we computed the cosine of the angle

$$\cos\alpha(t) = \frac{\|\mathbf{x}^{*T}\mathbf{x}(t)\|_2}{\|\mathbf{x}^*\|_2\|\mathbf{x}(t)\|_2}, \tag{4.4}$$

and used the angle $\alpha(t)$ as a metric, which tends to zero as t increases. As a second metric, we measured the maximum of the vector elements to verify normalization.

The failure scenarios involved varying message drop probabilities, and varying message delays. Message drop was modeled by dropping all messages with a given probability, and message delay and delay jitter was modeled by drawing a delay value from a specified interval uniformly, for all messages. Obviously, these settings were applied for all messages sent by any of the components of the protocol equally.

Figure 4.1 shows the results of the experiments. First of all, even the more moderate failure scenario can be considered pessimistic, not to mention the more severe scenario. This is because in the application scenarios we envision, the interval Δ can be rather long, in the range of ten to thirty seconds, so a delay of 10% of Δ is already large. Most

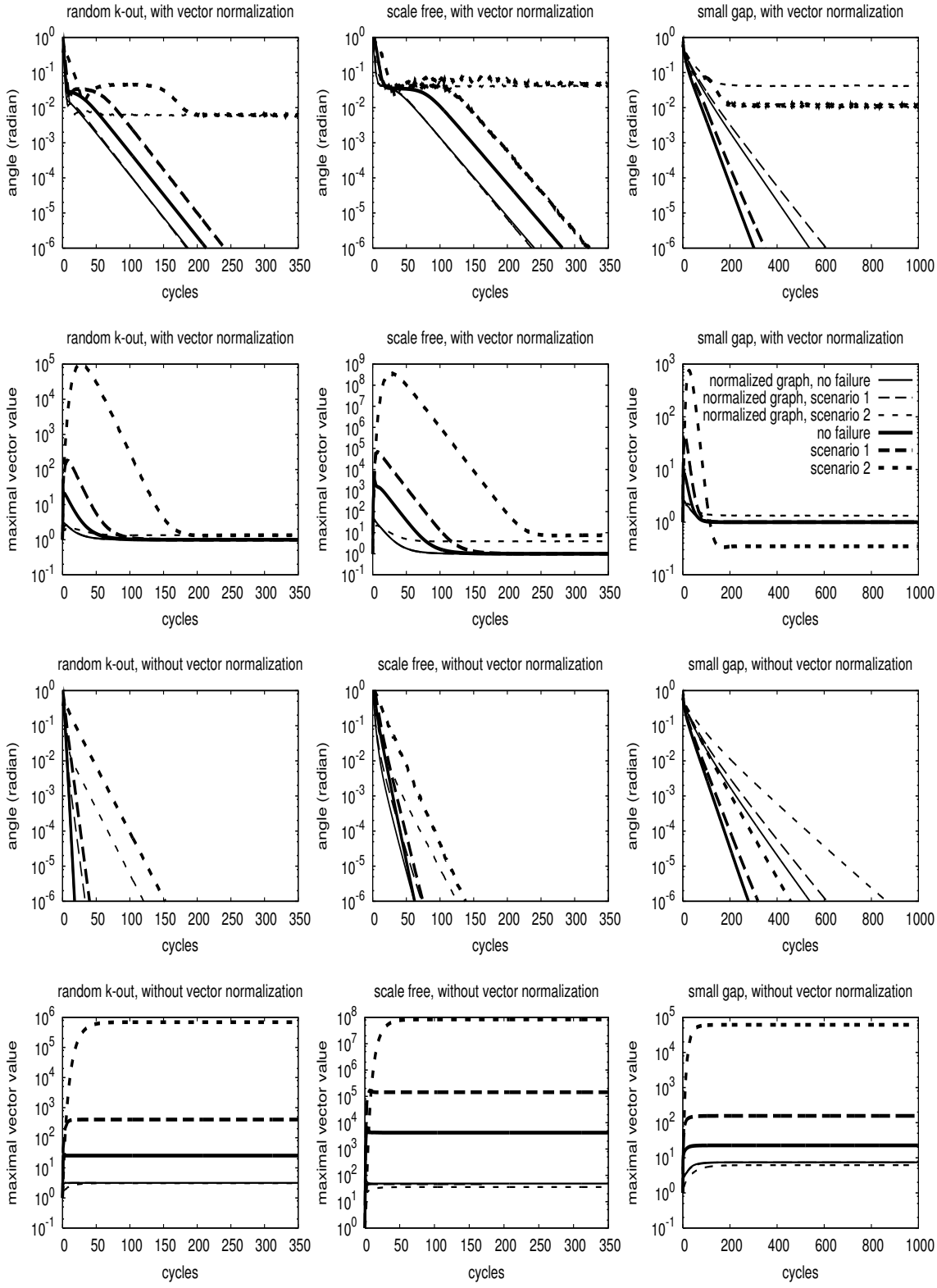


Figure 4.1: Simulation results. Scenario 1 involves $P_{drop} = 0.1$, and a random message delay drawn from $[0, \Delta/10]$ uniformly. In scenario 2, $P_{drop} = 0.3$ and the message delay is drawn from $[\Delta/10, \Delta/2]$.

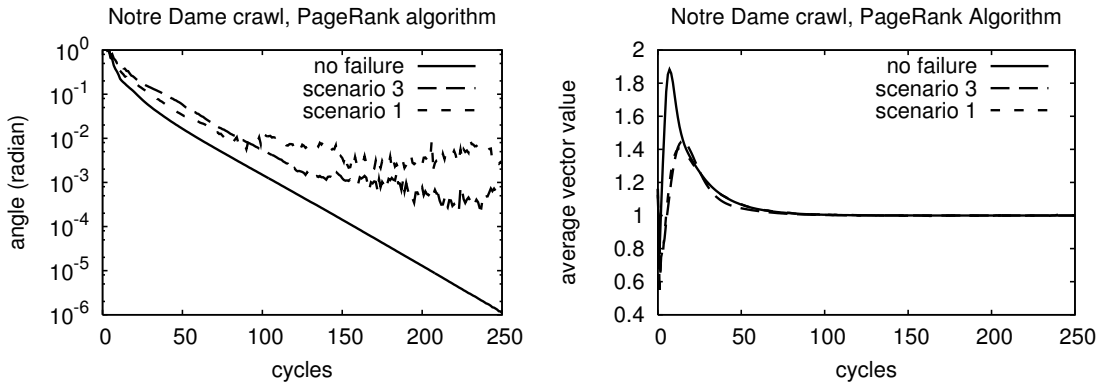


Figure 4.2: Simulation results with the PageRank algorithm. Scenario 1 involves $P_{drop} = 0.1$, and a random message delay drawn from $[0, \Delta/10]$ uniformly. In scenario 3, $P_{drop} = 0$ and the message delay is the same as in scenario 1.

importantly, from the point of view of the averaging and maximum finding protocols, that have a much shorter cycle length of $\Delta_n = \Delta/30$, these delay values are extreme.

From the experiments we can conclude that when the vector norm is not controlled explicitly, then convergence is fast, comparable to that of the centralized power iteration. Our preliminary experiments (not shown) suggest that message delay has virtually no effect on the convergence results, when $P_{drop} = 0$. Higher drop rates slow down convergence but do not change its characteristics significantly.

When we do apply vector normalization, convergence slows down somewhat due to the interference of vector normalization and asynchronous iteration. In the extreme failure scenario we don't achieve full convergence. The reason is that the extremely high delay and message drop rate prevents the propagation of the current maximum of the vector to all nodes during the interval Δ, and so different nodes might normalize with a different value. As a side effect, the maximum does not converge to one, and there is a constant noise factor in the approximation of the eigenvector. However, in the less severe, but still pessimistic scenario we do achieve convergence.

4.5.4 PageRank on WWW Crawl Data

As a more realistic case study, we tested our method on the same dataset used in [116], available from the authors. It was generated by a crawler, starting from one page within the domain of the University of Notre Dame. This sample has 325729 nodes. On this dataset we executed the PageRank algorithm, as described in Section 4.4. The weight of the random surfer operator was $\epsilon = 0.2$. This way, for the complete linear operator of the PageRank algorithm, we have $\lambda_1 = 0.84648, \lambda_2 = 0.8$.

The results of the method are shown in Figure 4.2 in various failure scenarios. We can observe that the protocol is now more sensitive to failure than in the case of the previous experiments, although the achieved accuracy is still satisfactory (note the logarithmic scale of the plots). The reason is that to get correct rank values the vector average must be used for controlling the norm of the vector, that is, it is guaranteed that the average of the vector stays one. The average is used to implement the random surfer operator as well. However, the calculation of the average is more sensitive to failure than the calculation of the maximum. This way, the approximation of the actual average of the vector has a small noise factor, that is inherited by the approximation of the ranks.

We can also note that the protocol scales well: the network examined here is two orders of magnitude larger than the previously examined networks, while convergence speed is still similar.

4.6 Related Work

Due to its importance, the distributed calculation of the dominant eigenvalue of large matrices has an extensive literature. In the area of parallel and cluster computing, the focus has largely been the optimization of existing, often iterative, methods on parallel computers and clusters (for a summary, see [115]). Such optimizations include partitioning; for example, different parts of the vector can be freely assigned to different processors in order to minimize message exchange and to maximize speedup. Besides, due to the reliable computing platform, synchronization can be efficiently implemented. This model is radically different from ours: in our case the assignment is fixed and given *a priory*, and the main goal is to achieve robustness to high rates of message delivery failures.

Asynchronous protocols have also been proposed for implementing iterative methods, and important convergence results are available as well (see [117] for a summary). These protocols are extremely fault tolerant and also efficient, but so far no algorithms are known that can deal with the case when the dominant eigenvalue is different from one. This introduces a certain sensitivity to dynamic environments even if $\lambda_1 \approx 1$, besides, many interesting applications where $\lambda_1 \neq 1$ cannot be tackled, for example [113].

Finally, in the context of P2P systems the main focus is on distributed PageRank implementations, where in all cases $\lambda_1 = 1$ is assumed, for example, [106–108]. The EigenTrust protocol in [109] also applies a similar implementation, but the authors assume all values are updated in each round, presumably unaware of the advantages of the long existing asynchronous version of the protocol, and thereby offering a rather fragile algorithm.

4.7 Conclusions

In this chapter we have addressed the problem of designing a fully distributed and robust algorithm for finding the dominant eigenvector of large and sparse matrices, that are represented as weights of links between nodes of a network. Our contribution can be summarized as follows. First of all, our algorithm does not require the dominant eigenvalue to be one. This is an important feature even if the problem involves a dominant eigenvalue of one (like PageRank does). In PageRank, sophisticated techniques for "fixing" the graph are required to make sure the dominant eigenvalue is one, which are not needed in our case, as we demonstrated. Besides, the protocol opens the door for applications where the dominant eigenvalue is known to be different from one [113].

Second, the norm of the approximation of the dominant eigenvector can be controlled as well. In other words, in addition to guaranteeing that the norm of the vector converges to a finite value, we can define this value explicitly using an additional gossip-component. This also means that the algorithm can be run indefinitely in a continuously changing environment.

Finally, we demonstrated the robustness of the algorithm through event-based simulation experiments, both on artificially generated graphs and on web-crawl data.

Chapter 5

Slicing Overlay Networks

In this chapter we demonstrate yet another application of the gossip scheme: we will show how to apply the gossip framework to implement a form of resource sharing via maintaining partitions in the network in the face of node churn and failures. An interesting aspect of this application is that—although it is seemingly unrelated to averaging at first sight—its convergence can be described with the same tools we developed in Chapter 3 to characterize the convergence of averaging.

The motivation of slicing is that recently there has been an increasing interest to harness the potential of P2P technology to design and build rich environments where services are provided and multiple applications can be supported in a flexible and dynamic manner. In such a context, resource assignment to services and applications is crucial. Current approaches require significant "manual-mode" operations and/or rely on centralized servers to maintain resource availability. Such approaches are neither scalable nor robust enough.

Our contribution towards the solution of this problem is proposing and evaluating a gossip-based protocol to automatically partition the available nodes into "slices", also taking into account specific attributes of the nodes. These slices can be assigned to run services or applications in a fully self-organizing but controlled manner. The main advantages of the proposed protocol are extreme scalability and robustness. We present approximative theoretical models and extensive empirical analysis of the proposed protocol.

5.1 Introduction

Following the scale shift in distributed systems and their increasing dynamism, peer-to-peer overlay networks have imposed themselves as the key to build and maintain large-scale dynamic distributed systems. One important problem in the field of overlay networks is the design of infrastructures on which several applications might run together and share resources. Examples of such applications are Desktop-grid like computing platforms [118], and testbed platforms such as PlanetLab [95].

One key sub-problem is such environments is resource assignment to services and applications, and the definition of the resource itself. For example, in PlanetLab, the core concept is a slice, which refers to a virtualized network running over multiple physical nodes, and where each node can participate in multiple slices. Such slices are assigned to specific applications, sharing the platform. However, existing approaches are mostly manual and/or centralized. In contrast to this, we are interested in massively large scale and extremely dynamic networks, in which centralized slice assignment is not an option

and where slices need not only to be assigned, but also *maintained*, to face constant churn.

In this chapter, as a step towards a full self-organizing architecture, we focus on a well-defined problem: *ordered slicing*. Our objective is to create and maintain a partitioning of the network (we call the partitions *slices* in the following). This implies that slices are defined as subsets of the network, that is, each node belongs to exactly one slice at any given point in time. However, several such partitionings can be maintained in parallel. The ordered nature of the slicing means that specific attributes can be taken into account to partition the network: the partitioning is done along a fixed attribute of the nodes. For example, a service might require a slice composed of the top 20% of the nodes providing the largest bandwidth. Besides, we need to provide this top 20% constantly, even if the nodes in the top 20% constantly change due to churn or changing node properties.

Many metrics may be used to sort the nodes such as available resources (memory, bandwidth, computing power) or some specific behavioral pattern such as up-time. Note that slicing the network at random, and focusing only on the size of the slices is a special case of our ordered slicing protocol. We also note that the slice sizes are expressed as a percentage of the network, that is, if the network grows, slices grow accordingly.

The rest of the chapter is organized as follows. In Section 5.2 we provide the problem statement and the system model. In Section 5.3 we describe our gossip-based slicing protocol. An approximative theoretical model of our approach is presented in Section 5.4 and an extensive empirical analysis is presented in Section 5.5.

5.2 Problem Definition

5.2.1 System Model

We consider a network consisting of a large collection of *nodes* that are assigned unique identifiers (typically IP addresses) and that communicate through message exchanges. The network is highly dynamic; new nodes may join at any time, and existing nodes may leave, either voluntarily or by *crashing*. In the following, we limit our discussion to node crashes. Voluntary leaves are implemented as crashes: our protocols will not require a dedicated leave procedure, nor any failure detection. Successful delivery of messages happens without delay, however, messages may be dropped. Byzantine failures, with nodes behaving arbitrarily, are excluded from the present discussion.

We assume that nodes are connected through an existing physical routed network, such as the Internet, where every node can potentially communicate with every other node. To actually communicate, a node has to know the identifiers of a set of other nodes (its *neighbors*), for example, the IP address in the case of an IP network. This neighborhood relation over the nodes defines the topology of the *overlay network*. Given the large scale and the dynamism of our envisioned system, neighborhoods are typically limited to small subsets of the entire network. The neighbors of a node (and, thus, the overlay topology) may change dynamically over time.

5.2.2 The Ordered Slicing Problem

Intuitively, the ordered slicing problem asks for a partitioning of the nodes in the overlay network into groups (slices) in such a way, that the groups are ordered with respect to some given metric, such as the availability of a resource, or some other relevant property. For example, we might be interested in creating and maintaining a slice composed of the

top 10% nodes according to available bandwidth, expected up-time, and so on. Note that creating a slice of a given size, populated with random nodes, is a special case where the metric is not taken into account, or, equivalently, assuming all nodes have the same value of the metric. Slice sizes are expressed as a percentage of the network size.

To define this problem, let N denote the network size and let each node i have an attribute, x_i. This value will typically measure the availability of some resource at node i. We assume that there exists a total ordering over the domain of the attributes values, so that the values in the network $(x_1, \ldots, x_N)$ can be ordered. Let us also assume that there is a *slice specification* that defines an ordered partitioning of the nodes. That is, the slice specification is a list of positive real numbers $s_1, \ldots, s_k$ such that $\sum s_i = 1$, that define slices $S_1, \ldots, S_k$, where the size of S_i is $s_i N$ and for all $i < j$, $a \in S_i$ and $b \in S_j$ we have $x_a \leq x_b$. We also assume that the slice specification is known at each node locally.

The problem is to automatically assign each node to slices in such a way that satisfies the slice specification, using only local message exchange with currently known neighbors. That is, we want each node to find out, which slice it belongs to, and, as a function of continuous changes in the network, maintain this assignment up-to-date.

The difficulty lies in the fact that the correct solution needs global information in that each node needs to calculate the number of nodes that preceed them in the total order, and break ties whenever the number of preceeding nodes is not well defined (for example, if there are many identical attribute values in the network). Furthermore, the dynamism and failures in the system add extra difficulty, as this assignment needs to be continuously maintained in the face of a changing set of nodes.

As opposed to most traditional approaches that require only eventual correctness provided there is no failure or change in the system for a sufficiently long time, we focus on a best effort approximation which is as close to the optimal solution as possible. In other words, instead of focusing on the worst case, we focus on optimizing the performance under normal dynamic operation. Nevertheless our solution is actually eventually consistent.

Note that the solution we present here can easily be extended to more general cases, such as when more independent attributes are involved, or when overlapping groups need to be maintained, and so on. However, the problem definition above allows us to keep the focus on the analysis of the key novel contributions introduced here.

5.3 A Gossip-based Approach

As mentioned previously, to let nodes decide locally whether they belong to a certain slice or not (expressed at a percentage of the whole size which is not known either), the key issue is to enable a node to approximate what percentage of nodes preceed it in the ordering according to the attribute value. There are at least two natural choices to implement this functionality. The first is through the application of protocols to calculate the ranking of the nodes in the ordering [13,34]. However, known protocols are expensive and they are not suitable to *maintain* ranking information cheaply in the face of large scale and dynamism. The second is to approximate the distribution of the attribute values and use this information to map any attribute value to an approximate ranking [119,120]. This approach however is not robust to skewed distributions and does not provide a sufficiently accurate information for the present purposes.

To deal with dynamism and large scale, we follow a third approach, which is based on the *sorting* of randomly generated numbers. The basic idea is that each node generates one

Algorithm 12 Slicing

```
1: loop
2:    wait(Δ)
3:    p ← selectSlicePeer()
4:    if p ≠ null then
5:       sendPush(p,(x, r))
6: procedure ONPUSH(m)
7:    sendPull(m.sender,(x, r))
8:    onPull(m)
9:
10: procedure ONPULL(m)
11:    if (x − m.x)(r − m.r) < 0 then
12:       r ← m.r
```

uniform random number from a fixed interval, and subsequently the set of these random numbers are sorted "along the attribute values" with the help of the protocol we describe below. Sorting along the attribute values means that—via swapping the random numbers among a suitable sequence of pairs of nodes—we would like to achieve that the order of the random numbers reflects the order of the attribute values over the nodes.

After sorting, the node is able to make a judgment about its position in the sorting of the attribute values based on the random number it currently holds, because the distribution of the random numbers is known (that is, uniform from a fixed interval) and because the order of the random numbers reflect the order of the attribute values. For example, if the random numbers are drawn from $[0, 1]$, then a node decides that it is in the first half of the sorting if, after sorting along the attribute values, it holds a value less than 0.5. Apart from being simple, this approach supports dynamism well, as all joining nodes can locally initialize their random number and subsequently participate in the sorting. Furthermore, the approach works independently of the distribution of the attribute values: they can even be identical at all nodes, in which case only the sizes of the slices are determined, but the nodes will be assigned to slices at random.

However, the sorting problem might seem equally difficult to our original problem. Our main contribution—apart from proposing the application of sorting—is a gossip-based sorting protocol that is simple to implement, incurs minimal costs and is efficient enough for the purposes of ordered slicing. The basic idea relies on a simple swapping of the random numbers between nodes. For example, let nodes i and j have attribute values $x_i = 10$ and $x_j = 20$, and random numbers $r_i = 0.8$ and $r_j = 0.1$. These nodes simply swap their random numbers in order to make them reflect the ordering of the attribute value. In order to make such pairs of nodes discover each other, we rely on a gossip-based algorithm.

Our sorting protocol is based on the NEWSCAST protocol (see Chapter 2.2.4). The basic idea behind the sorting algorithm is that each node will passively look for candidate peers to swap its random value with, in order to improve the sorting. These candidates are discovered using the constantly changing set of neighbors provided by the NEWSCAST protocol. The sorting protocol will be based on NEWSCAST not through the usual peer sampling API (method SELECTPEER that returns a random peer) but it will have a more intimate relationship with NEWSCAST. First, the node descriptors in the NEWSCAST view will contain not only the node address and the timestamp, but also the values of x and r at the node at the time of submitting the descriptor. Second, the slicing protocol will ask for peers from the newscast view that are likely to be good candidates to swap random values with.

The algorithm is shown in Algorithm 12. On peer i method SELECTSLICEPEER returns a peer j from the NEWSCAST view such that $(x_i - x_j)(r_i - r_j) < 0$, which means that the given peer is a potential candidate to swap random values with. It is not guaranteed that

a suitable peer exists in the view. If there is no suitable peer then no push message is sent and therefore no exchange is performed.

Note, that if $(x_i - x_j)(r_i - r_j) \geq 0$ according to the current view, it is still possible that in reality peer j has become a good candidate in the meantime, because the relative order of two peers can potentially be reversed and the information in the descriptor might be slightly outdated. Similarly, it is possible that—although node j seems to be a suitable one—in the meantime its random value has changed and it is not actually suitable anymore. In this latter case the push and pull messages are sent (in vain) but no exchange happens.

5.4 Analogy with Gossip-based Averaging

In this section, we analyze the protocol mostly based on the insight that the protocol can be considered an instance of gossip based averaging (see Chapter 3). For the present section, we assume here that the attribute values do not change and that the set of nodes does not change either. This also means that the set of random numbers r_i held by the nodes remains the same at any given point in time.

Let us define $\rho(x_i)$ to be the rank of x_i in the sorting of the x values, and, similarly, let $\rho(r_i)$ be the rank of r_i in the sorting of the r values. Let

$$\delta_i(t) = \rho(x_i(t)) - \rho(r_i(t)). \tag{5.1}$$

When $\delta_i(t) = 0$, we know that $r_i(t)$ is the random value that belongs to node i. The state we want to reach is the one in which $\delta_i = 0$ for all $i = 1, \ldots, N$, at which point the r_i values are sorted w.r.t. the x_i values. Let us assume that all the values x_i and r_i are unique, so their rank is well defined. This is purely to keep our discussion simple. Note that this is the worst case: if some of the values are not unique, then the problem becomes easier; for example, if all the values r_i or x_i are the same, then all permutations are sorted and there is no problem to solve.

Our main observation is that the slicing protocol in Algorithm 12 can be considered an averaging algorithm over the values δ_i. To see this, consider first that the sum of these values remains zero after each exchange. That is, we have the mass conservation property:

$$\sum_{j=1}^{N} \delta_i(t) = \sum_{j=1}^{N} \rho(x_i(t)) - \rho(r_i(t)) = \sum_{j=1}^{N} \rho(x_i(t)) - \sum_{j=1}^{N} \rho(r_i(t)) = 0 \tag{5.2}$$

Similarly to averaging, it is easy to verify that the maximal δ value will decrease every time its node participates in a successful exchange. Similarly, the minimal value will increase. Besides, it is evident that for each node, participating in a successful exchange has a finite probability in each cycle. This means that the variance of the values decreases, which proves convergence in probability. The following proposition will describe the effect of an exchange on the participating δ values in more detail.

Proposition 5.4.1. *Let node i and j perform a successful exchange at time t. After the exchange, we have*

$$E(\delta_i(t+1)) = E(\delta_j(t+1)) = \frac{\delta_i(t) + \delta_j(t)}{2} \tag{5.3}$$

Proof. Since the exchange was successful, we must have $(x_i(t)-x_j(t))(r_i(t)-r_i(t)) < 0$. Actually, this also means that

$$(\rho(x_i(t)) - \rho(x_j(t)))(\rho(r_i(t)) - \rho(r_i(t))) < 0. \tag{5.4}$$

Without loss of generality, let us assume throughout the proof that $\delta_i(t) < 0$ and $|\delta_i(t)| > |\delta_j(t)|$ (the other logical cases are symmetrical to this). In this case, the possible values of $\rho(r_j(t))$ consistent with (5.4) are $\rho(r_i(t))-1, \rho(r_i(t))-2, \ldots, \rho(x_i(t))-\delta_j(t)+1$. Note that since we know $\delta_j(t)$, we know that $\rho(x_j(t)) = \rho(r_j(t)) + \delta_j(t)$. The $(\delta_i(t+1), \delta_j(t+1))$ pairs that belong to these options are $(\delta_i(t)+1, \delta_j(t)-1)$, $(\delta_i(t)+2, \delta_j(t)-2), \ldots, (\delta_j(t)-1, \delta_i(t)+1)$.

Now, we know that $1 \leq \rho(x_i(t)) \leq N + \delta_i(t)$. For any fixed value of $\rho(x_i(t))$ such that $1 \leq \rho(x_i(t)) - \delta_j(t) + 1$ and $\rho(r_i(t)) + \delta_j(t) - 1 \leq N$, we know that all options listed above are possible and have equal probability, so

$$E(\delta_i(t+1) \mid \rho(x_i(t)) = E(\delta_j(t+1) \mid \rho(x_i(t)) = \\ \frac{1}{n}\sum_{k=1}^{n} \delta_i(t) + k = \frac{1}{n}\frac{n(\delta_i(t)+\delta_j(t))}{2} = \frac{\delta_i(t)+\delta_j(t)}{2}, \tag{5.5}$$

where $n = \delta_j(t) - \delta_i(t) - 1$.

We only sketch how to deal with the case when the value of $\rho(x_i(t))$ is such that $1 > \rho(x_i(t)) - \delta_j(t) + 1$ or $\rho(r_i(t)) + \delta_j(t) - 1 > N$. In this case, some of the options are not possible. However, these impossible options are symmetric: if some settings for node j are not possible because $\rho(x_i(t))$ is too close to 1, then we have a symmetric value of $\rho(x_i(t))$ (too close to N) where the same number of options are not possible on the opposite end of the list of options. Considering the symmetry of the series of options for $(\delta_i(t+1), \delta_j(t+1))$, it is not hard to see that the the desired expectation holds also in this regime. □

This result makes slicing very similar to averaging, since the only difference is that in the case of averaging both nodes will have exactly the average $(x_i(t) + x_j(t))/2$, whereas here this is true only in expectation.

Like in Chapter 3, to characterize the convergence of the protocol we will focus on the variance of the values that here we will call the *disorder measure*:

$$\sigma(t) = \frac{1}{N}\sum_{i=1}^{N} \delta_i(t)^2, \tag{5.6}$$

This measure is minimized when the sorting is perfect, when it takes the value of zero. The value of this measure is shown in Figure 5.1. Note that the figure indicates that less than 20 cycles are sufficient to reduce the average error to 1% of the network size, which means that nodes are $N/100$ positions away from their correct position on average, independently of network size. With $c = 80$, 40 cycles are enough to reach 0.1% precision.

From Chapter 3 Proposition 3.3.5 and Corollary 3.3.3 can be shown to hold in the case of slicing as well using Proposition 5.4.1. Therefore we expect that the disorder is reduced exponentially fast as a function of the number of successful exchanges (note that in the case of averaging every peer can perform a successful exchange with every other peer at any time, whereas here it becomes harder and harder to find suitable peers). As can be seen in Figure 5.2, the qualitative prediction of exponential behavior is very accurate.

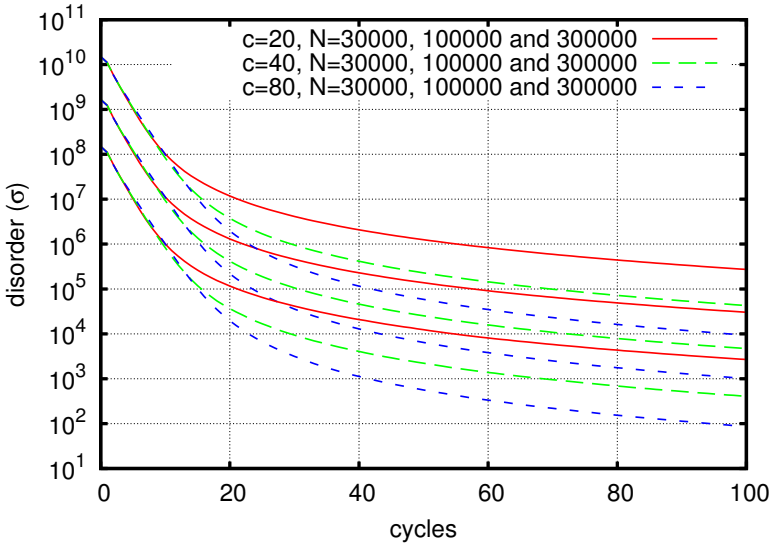


Figure 5.1: Disorder as a function of cycles. Curves for a fixed c completely overlap when normalized by N^2.

Furthermore, slicing appears to approximate the theoretically minimal convergence factor, that is, $a^* = 1/4$ (using the notation from Proposition 3.3.5 and Corollary 3.3.3). In the case of averaging, this minimal convergence factor could be reached using a series of exchanges based on two independent perfect matchings in one cycle. In the case of slicing, we speculate that any series of $N/2$ exchanges are closer to a perfect matching than to a set of random pairs. This is because in each cycle the set of suitable peers for any node is relatively small, and its size is proportional to $\delta_i(t)$, given that $\delta_i(t)$ at node i decreases at roughly the same rate for each node i.

5.5 Experimental Analysis

We have performed extensive simulation experiments in order to study the probability of finding a peer to swap with, as well as the behavior of the protocol in the presence of message drop and node dynamism (churn). The experiments were performed using the PEERSIM simulator [66]. All scenarios were run with three network sizes ($N = 30000$, 100000 and 300000) and three view size settings ($c = 20$, 40 and 80).

5.5.1 The Number of Successful Swaps

Since the nodes are not guaranteed to find suitable peers to swap values with, the exponential convergence is valid only as a function of the number of successful swaps, and not as a function of cycles. Here we experimentally evaluate the probability of finding a suitable peer as a function of time.

Figure 5.3 shows our experimental results regarding the number of successful swaps as a function of cycles. The number of swaps depends on the view size parameter c, but in all cases it has a power-law tail that decreases approximately as $1/x$. In the first cycles however, the number of exchanges remain approximately constant. This is due to the fact that the algorithm uses $c > 1$ candidates to eventually select a peer. If $p(t)$ is the probability that a random peer is a suitable peer, then in each selection step, the algorithm

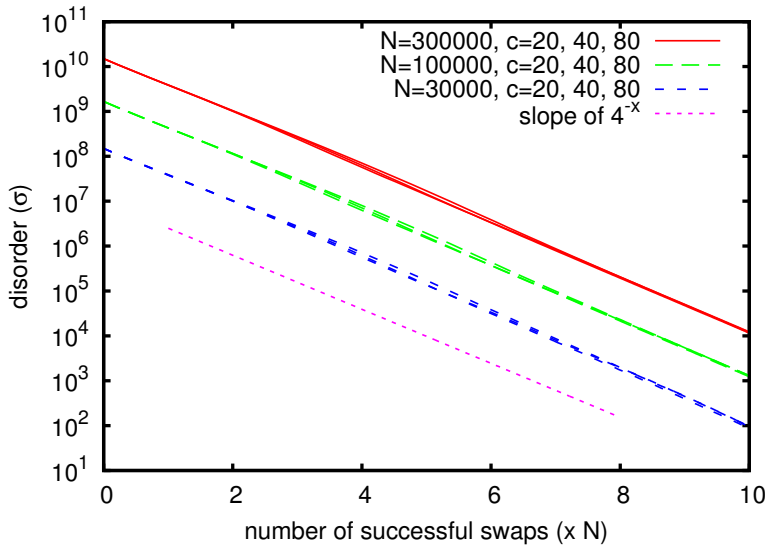


Figure 5.2: The exponential decrease of the disorder as a function of number of successful swaps (normalized by the network size), for different values of parameter c (view size) and network sizes. Lines that belong to the same network size fully overlap.

selects a suitable peer with probability $1 - (1 - p(t))^c$. While $p(t)$ is large (that is, while t is small), this probability remains close to one, and as a result the disorder σ decreases exponentially fast as a function of cycles (see Figure 5.1). However, when $p(t)$ becomes small (as $\delta_i(t)$ becomes small), convergence slows down. Most importantly, this result is independent of network size which allows for a scalable and robust setting of parameters.

5.5.2 Message Drop

The protocol generates a large number of independent message exchanges (push and pull) at all nodes. In the implementation, the messages are assumed to be sent using an unreliable channel, such as UDP, and there is no failure detection mechanism.

If the push message is dropped, the exchange is dropped as a whole. These cases simply slow down the convergence proportionally to the number of failures, without changing its characteristics.

If the pull message is dropped then the random value originally held by the selected peer is lost, since the selected peer first sets the value received in the push message. In other words, one of the values gets duplicated and the other gets lost. This however has no dramatic effect, as long as there are still a sufficient number of different values, since the distribution of the set of all values is still uniform (since no bias is involved in the message failures). Indeed, as shown in Figure 5.4, we can only observe a proportional slowdown, under 10% uniform message drop, that can be considered a rather significant drop rate. For other network sizes we obtain identical results. We can conclude that the quality of ordering is highly robust to message drop failures.

However, diversity of the values is important, because the "resolution" of the system (the number and size of the groups it can order) depends on this diversity. If there is no churn, then there will be fewer and fewer swaps as the system converges to the ordered state, as described in Section 5.4. Besides, when each value still represented has a small number of copies, it becomes very unlikely that all copies of a specific value are completely removed. Due to these two properties, diversity practically stops decreasing very

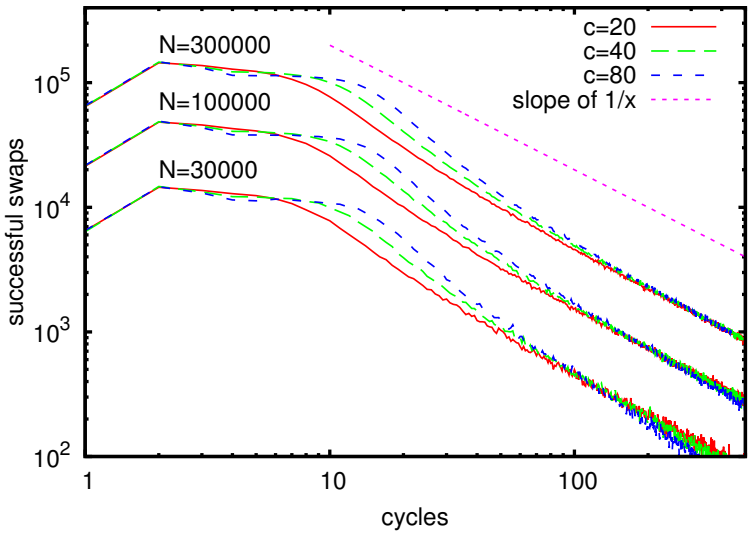


Figure 5.3: Swaps as a function of cycles. Curves completely overlap when normalized by N.

soon. In addition, in the presence of extreme failure rates, we can add a simple technique to further fight the lack of diversity: whenever a node sees another node in its view that holds the same random value, it replaces its own value with a random one. This technique in effect introduces a very low rate artificial churn, that is dealt with just like real churn (see Section 5.5.3). We also note that if there is natural churn, then diversity is maintained by the churn itself.

5.5.3 Churn

To examine the effect of churn, we define an artificial scenario in which a given proportion of the nodes crash and are subsequently replaced by new nodes in each cycle. This scenario is a *worst case* scenario because the new nodes are assumed to join the system for the first time (their random value r_i is independent of their attribute value x_i) and the crashed nodes are assumed never to join the system again. The view of joining nodes is initialized with descriptors of randomly selected nodes.

Churn rate defines the number of nodes that are replaced by new nodes in each cycle. We consider the churn rates 0.1% and 1%. Since churn is defined in terms of cycles, in order to validate how realistic these settings are, we need to define the cycle length. With the very conservative setting of 10 seconds, which results in a very low load at each node, the trace described in [58] corresponds to 0.2% churn in each cycle. In this light, we consider 1% a comfortable upper bound of realistic churn, given also that the cycle length can easily be decreased as well to deal with even higher levels of churn.

The results of the experiments are shown in Figure 5.5. The ordering effort of the protocol and the continuously introduced disorder reaches an equilibrium after a few cycles, after which the level of order remains stable. Even with a 1% churn rate in each cycle, the protocol manages to keep the average distance from the correct position by approximately an order of magnitude less than that in a random permutation. Note that in this scenario, during the 50 cycles shown, almost half of the network gets replaced at least once. We can further improve the performance of the protocol using techniques that take the age (time spent in the network) into account. One technique is called *age bias*; when using

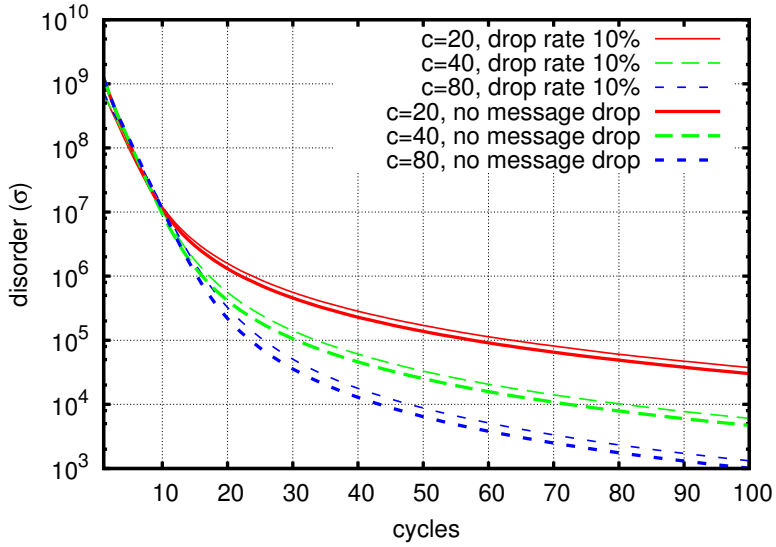


Figure 5.4: Disorder as a function of cycles, with and without message drop, for $N = 100000$.

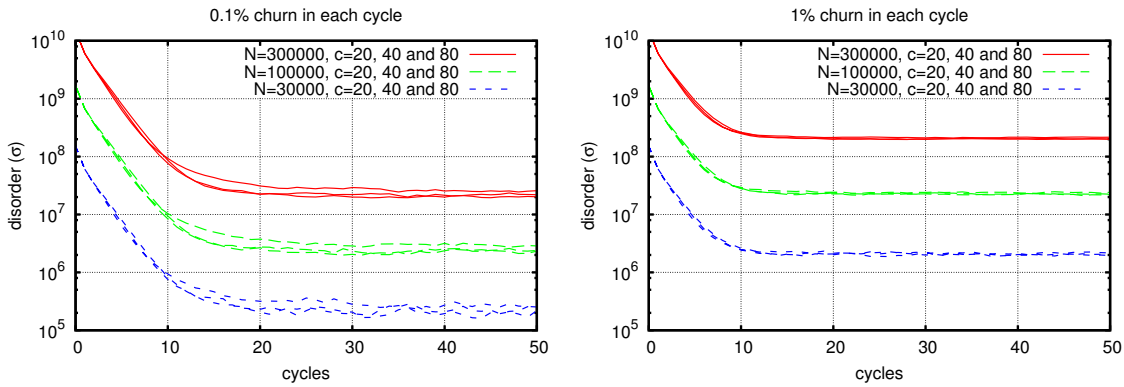


Figure 5.5: Disorder as a function of cycles, for churn rates 0.1% and 1% per cycle. Curves completely overlap when normalized by N^2.

this technique, a node, when selecting the neighbor to swap with, chooses the one among the candidates which has the most similar age. This can be easily implemented without extra communication steps, if the node descriptors in the view also contain node age. As a result, only the younger nodes tend to be disordered, while they can still converge and while the older nodes that have already converged remain protected. Indeed, as shown in Figure 5.6, we obtain a considerable improvement using the age bias technique, if, in addition to the age bias, we also require a certain maturity (that is, minimal age) to be considered as part of any slice. In other words, the order among the nodes that have a certain minimal age improves significantly.

5.5.4 An Illustrative Example

To illustrate how well our approach copes well with highly dynamic environments, Figure 5.7 provides a visualization of three slices that are maintained in a network of size 1000, over 1200 cycles, using age bias and a maturity level of 20 cycle. The slice specification is $(1/3, 1/3, 1/3)$, we have three slices of equal size. The view size is $c = 20$.

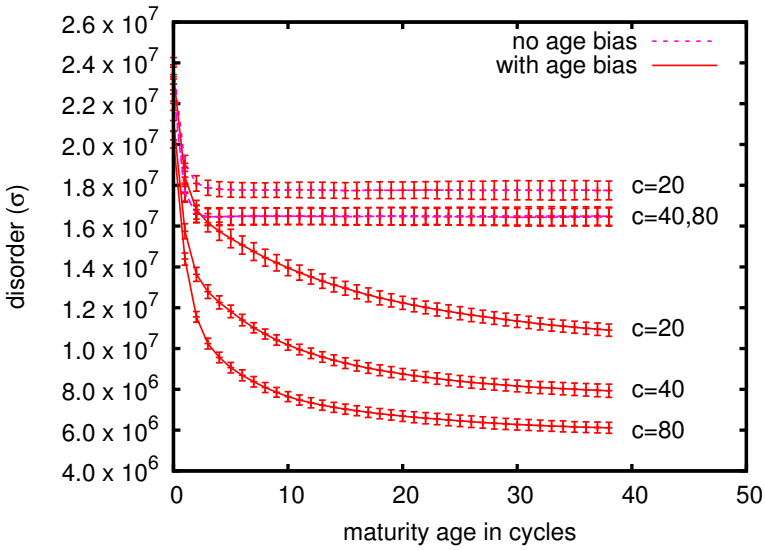


Figure 5.6: Effects of age-based techniques. The converged value of σ is shown. Network size is 100000, error bars show standard deviation over 50 cycles (cycles 50–100).

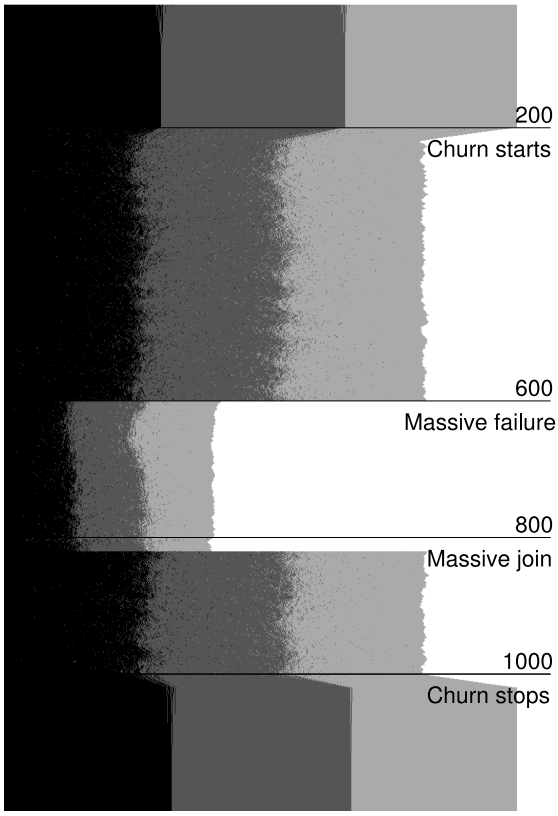


Figure 5.7: Visualization of groups in extreme failure scenarios.

After the start of churn the network seems to shrink. This is due to the fact that we consider only mature nodes, that is, those that are older than 20 cycles.. The scenario we applied includes churn (1% in each cycle), removal of a random half of the network and subsequently duplicating network size in one cycle. We observe that the slices remain relatively well defined, especially if we consider that the entire network gets replaced several times during the period shown. We also observe that as soon as the churn stops, the slices get stabilized as well. Note that our goal cannot be to eliminate churn within a slice completely, but only to make sure it is similar to the churn the entire network is experiencing.

5.6 Conclusions

We have described a solution to automatically partition a highly dynamic network according to a given metric as well as to maintain such a partitioning in the presence of churn. In our approach to the ordered slicing problem, each node has to identify which section of the network it belongs to, ordered along an attribute x_i, using only local information. Our solution relies on a robust and scalable gossip-based sorting protocol. We have presented approximative theoretical results based on an analogy with average calculation and demonstrated the robustness of the protocol in simulation experiments.

We focused only on the identification of the slices, which is in itself a challenging problem. However, to be practically useful, these slices have to be presented to users and applications as groups. One solution to this problem is to execute a slice specific NEWSCAST protocol inside each slice, which implements the peer sampling service (providing samples from the slice). Users of a slice will simply be part of the slice and access it through the peer sampling service. Nodes (and users) can join a slice-specific NEWSCAST via a random contact from the slice. Such contacts can be stored (and continuously updated) together with the slice specification, which, as we mentioned previously, can be thought of as a very small database stored at each node and maintained cheaply using anti-entropy gossip.

Chapter 6

T-Man: Topology Construction

So far we have focused on computations based on the peer sampling service that is implemented using a random overlay network. Now we turn our attention to the problem of constructing structured overlay networks that can be used to implement distributed data structures or to optimize distributed applications via exploiting geographical node proximity or the similarity of interests of the participating agents.

In general, large-scale overlay networks have become crucial ingredients of fully-decentralized applications and peer-to-peer systems. Depending on the task at hand, overlay networks are organized into different topologies, such as rings, trees, semantic and geographic proximity networks. We argue that the central role overlay networks play in decentralized application development requires a more systematic study and effort towards understanding the possibilities and limits of overlay network construction in its generality.

The contribution of this chapter is a gossip protocol called T-MAN that can build a wide range of overlay networks from scratch, relying only on minimal assumptions. The protocol is fast, robust, and very simple. It is also highly configurable as the desired topology itself is a parameter in the form of a ranking method that orders nodes according to preference for a base node to select them as neighbors. We present extensive empirical analysis of the protocol along with theoretical analysis of certain aspects of its behavior.

6.1 Introduction

Overlay networks have emerged as perhaps the single-most important abstraction when implementing a wide range of functions in large, fully decentralized systems. The overlay network needs to be designed appropriately to support the application at hand efficiently. For example, application-level multicast might need carefully controlled random networks or trees, depending on the multicast approach [62, 87]. Similarly, decentralized search applications benefit from special overlay network structures such as random or scale-free graphs [121, 122], superpeer networks [123], networks that are organized based on proximity and/or capacity of the nodes [124, 125], or distributed hash tables (DHT-s), for example, [81, 83].

In current work, protocol designers typically assume that a given network exists for a long period of time, and only a relatively small proportion of nodes join or leave concurrently. Furthermore, applications either rely on their own idiosyncratic procedures for implementing join and repair of the overlay network or they simply let the network evolve in an emergent manner based on external factors such as user behavior.

We believe that there is room and need for interesting research contributions on at least two fronts. The first concerns the question whether a single framework can be used to develop flexible and configurable protocols without sacrificing simplicity and performance to tackle the plethora of overlay networks that have been proposed. The second front concerns scenarios in overlay construction that are often overlooked, such as massive joins and leaves, as well as quick and efficient bootstrapping of a desired overlay from scratch or some initial state. Current approaches either fail or are prohibitively expensive in such scenarios. Combining results on these two fronts would enable several interesting possibilities. These include: (i) overlay network creation *on demand*, (ii) deployment of temporary and adaptive decentralized applications with custom overlay topologies that are designed on-the-fly, (iii) federation or splitting of different existing architectures [17].

We address both questions and present an algorithm called T-MAN for creating a large class of overlay networks from scratch. The algorithm is highly configurable: the network to be created is defined compactly by a *ranking method*. The ranking method formalizes the following idea: when shown a set of nodes, we assume each node in the network is able to decide which ones it likes from the set more and which ones it likes less (we will later use this ability of nodes to help them have neighbors they like as much as possible). In other words, each node can order any set of nodes. Formally speaking, the ranking method is able to order any set of nodes given a so called *base node*. By defining an appropriate ranking method, we will be able to build a wide variety of topologies, including sorted rings, trees, toruses, clustering and proximity networks, and even full-blown DHT networks, such as the CHORD ring with fingers (as discussed in Chapter7). T-MAN relies only on an underlying peer sampling service (see Chapter 2) that creates an initial overlay network with random links as the starting point.

The algorithm is gossip based: all nodes periodically communicate with a randomly-selected neighbor and exchange (bounded) neighborhood information in order to improve the quality of their own neighbor set. This approach, while requiring no more messages than the heartbeats already present in proactive repair protocols, is simple, and achieves fast and robust convergence as we demonstrate. Here, we limit our study to the overlay construction problem. Our main contribution is to show that a single, generic gossip-based algorithm can *create* many different overlay networks *from scratch* quickly and efficiently.

The roadmap of the chapter is as follows. Sections 6.3 and 6.4 present the system model and the overlay construction problem. Section 6.5 describes the T-MAN protocol. In Section 6.6 we present theoretical and experimental results to characterize key properties of the protocol and to give guidelines on parameter settings. Section 6.7 presents practical extensions to the protocol related to bootstrapping and termination, and extensive experimental results are also given to examine the behavior of the protocol in different failure scenarios. Section 6.8 concludes the chapter.

6.2 Related Work and Contribution

Related work in bootstrapping include the algorithm of Voulgaris and van Steen [126] who propose a method to jump-start PASTRY [81]. This protocol is specifically tailored to PASTRY and its message complexity is significantly higher than that of T-MAN. More recently, the bootstrapping problem has been addressed in other specific overlays [127–129]. These algorithms, although reasonably efficient, are specific to their target overlay networks.

An approach closer to T-MAN is VICINITY, described in [130]. Although VICINITY was inspired by the earliest version of T-MAN, it does contain notable original components related to overlay maintenance, such as churn management, and other techniques to boost performance.

Finally, we mention related work that use gossip-based probabilistic and lightweight algorithms. We note that these algorithms are targeted neither at efficient bootstrapping, nor at generic topology management. Massoulié and Kermarrec [131] propose a protocol to evolve a topology that reflects proximity. More recent protocols applying similar principles include [132] and [133]. Repair protocols used extensively in many DHT overlays also belong to this category (e.g., [83, 134, 135]).

Our contribution with respect to related work is threefold. First, we introduce a lightweight probabilistic protocol that can construct a wide range of overlay networks based on a compact and intuitive representation: the ranking method. The protocol has a small number of parameters, and relies on minimal assumptions, such as nodes being able to obtain a random sample from the network (the peer sampling service). The protocol is an improved and simplified version of earlier variants presented at various workshops [15–17]. Second, we develop novel insights for the tradeoffs of parameter settings based on an analogy between T-MAN and epidemic broadcasts. We describe the dynamics of the protocol considering it as an epidemic broadcast, restricted by certain factors defined by the parameters and properties of the ranking method (that is, the properties of the desired overlay network). We also analyze storage complexity. Third, we present novel algorithmic techniques for initiating and terminating the protocol execution. We present extensive simulation results that support the efficiency and reliability of T-MAN.

6.3 System Model

We consider a set of nodes connected through a routed network. Each node has an address that is necessary and sufficient for sending it a message. Furthermore, all nodes have a *profile* containing any additional information about the node that is relevant for the definition of an overlay network. Node ID, geographical location, available resources, etc. are all examples of profile information. The address and the profile together form the *node descriptor*. At times, we will use "node descriptor" and "node" interchangeably if this does not cause confusion.

The network is highly dynamic; new nodes may join at any time and existing nodes may leave, either voluntarily or by *crashing*. Our approach does not require any mechanism specific to leaves: spontaneous crashes and voluntary leaves are treated uniformly. Thus, in the following, we limit our discussion to node crashes. Byzantine failures, with nodes behaving arbitrarily, are excluded from the present discussion.

We assume that nodes are connected through an existing routed network, where every node can potentially communicate with every other node. To actually communicate, a node has to know the address of the other node. This is achieved by maintaining a *partial view* (*view* for short) at each node that contains a set of node descriptors. Views can be interpreted as sets of edges between nodes, naturally defining a directed graph over the nodes that determines the topology of an *overlay network*.

Communication incurs unpredictable delays and may be subject to failures. Single messages could lost, links between pairs of nodes may break. Nodes have access to local clocks that can measure the passage of real time with reasonable accuracy, that is, with small short-term drift. Local clocks are not required to be synchronized.

Finally, we assume that all nodes have access to the peer sampling service (see Chapter 2) that returns random samples from the set of nodes in question. From a theoretical point of view we will assume that these samples are indeed random. From a practical point of view, we have seen that the peer sampling service indeed has suitable realistic implementations that provide high quality samples at a low cost.

6.4 The Overlay Construction Problem

Intuitively, we are interested in constructing some desirable overlay network, possibly from scratch, by filling the views at all nodes with descriptors of the appropriate neighbors. For example, we might want to organize the nodes into a ring where the nodes appear in increasing order based on their ID. Or we might want to construct a proximity network, where the neighbors of a node are those that are closest to it according to some metric.

We allow for arbitrary initial content of the views of the nodes in this problem definition (including empty views), noting that, as mentioned in our system model, nodes have access to random samples from the network, so they have access to at least random nodes from the network. In other words, starting from any arbitrary network, we want to fill the node views with the appropriate neighbors as fast as possible at a reasonable cost.

In order to have a well defined problem, we need to specify how the desired overlay is represented as an input to the protocol. The representation must be compact, intuitive, yet descriptive enough to capture the widest possible range of topologies.

Our proposal for the representing the desired overlay is the *ranking method*. As explained before, the ranking method sorts a set of nodes (potential neighbors) according to the "taste" of a given base node. More formally, the input of the problem is a set of N nodes, the *target view size* K (bounded by N) and a *ranking method* RANK. The ranking method takes as parameters the base node x and a set of nodes $\{y_1, \ldots, y_j\}$, $j \leq N$, and outputs an ordered list of these j nodes. All nodes in the network apply the same ranking method, which they are assumed to know a priori. We will analyze and test only ranking methods that are based on a partial ordering of the given set, and that return some total ordering consistent with this partial ordering (note however, that this is not an inherent restriction). Accordingly, we allow for an element of uncertainty (if there can be many total orderings consistent with the partial ordering we pick a random one).

A *target graph* that we wish to construct is defined by the ranking method. We present the definition of a target graph in a constructive way, through the following (inefficient) approach, for illustration. In this approach, each node disseminates its descriptor to all other nodes such that eventually, every node has collected locally the descriptor of every node in the network. At this point, each node sorts this set of descriptors according to the ranking method and picks the first K elements to be its neighbors. The resulting structure is called a *target graph*. Note that in this manner we define a graph, and not only a topology, because in addition to knowing the structure of the network, such as a ring, we also know the exact location of each node in the structure.

Disseminating all the descriptors to all the nodes is a naive solution to this *overlay construction problem* with a communication cost that is at least linear in N for each node and a storage cost that is also linear in N for each node. A practical approach has to significantly reduce the cost of this naive solution both in terms of communication and storage. The T-MAN protocol described in the next section achieves precisely this.

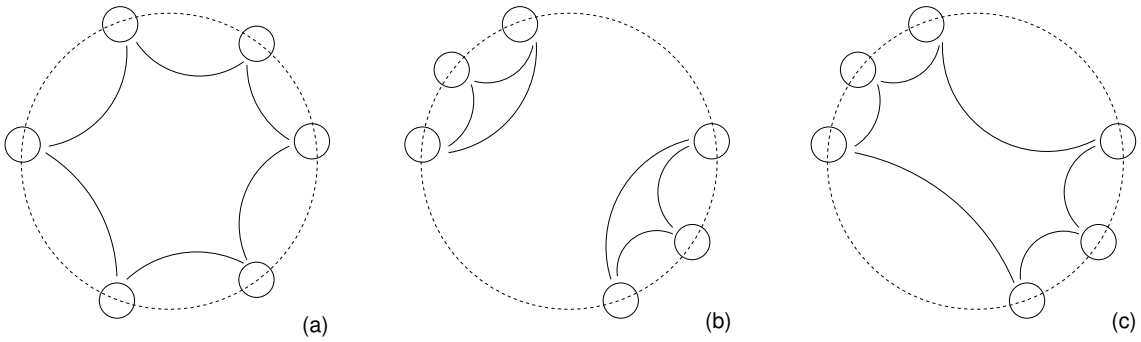


Figure 6.1: Target graphs for different ranking methods and $K = 2$. (a) One-dimensional distance-based, circular ranking method applied to a set of uniform node profiles; (b) same ranking method as before but with a different set of node profiles that are clustered; (c) direction-dependent ranking method achieves sorting even for clustered node profiles.

Although representing the target graph through the ranking method and parameter K clearly restricts the scope of the algorithm, through the examples presented here and in the rest of this chapter we will see that a wide range of interesting applications are covered. One (but not the only!) way of actually defining useful ranking methods is through a distance function that defines a metric space over the set of nodes. The ranking method can simply return an ordering of the given set according to non-decreasing distance from the base node.

To clarify the notions of ranking method and target graphs, let us consider a few simple examples, where $K = 2$ and the profile of a node is a real number in the interval $[0, M[$. We can define a ranking method based on the one-dimensional distance function $d(a, b) = |a - b|$, in which case the target graph will be linear. Alternatively—to connect the two ends of the line—we can use $d(a, b) = \min(M - |a - b|, |a - b|)$, which results in a circular structure. As illustrated in Figure 6.1(a), if the node profiles are more-or-less uniformly distributed over the interval $[0, M[$, the target graph that belongs to the circular distance function will be a connected ring. If the node profiles are not evenly distributed over $[0, M[$ but are clustered, the very same ranking method will result in a target graph that consists of disconnected clusters (Figure 6.1(b)).

It is important to note that there are target graphs of practical interest that cannot be defined through a global distance function. This is the main reason for using ranking methods, as opposed to relying exclusively on the notion of distance; the ranking method is a more general concept than distance. This fact will become important in Chapter 7 (practical application example), where it is necessary to be able to build, for example, a ring, even in the case of uneven node descriptor distributions when distance-based ranking methods would define clustered target graphs (as in Figure 6.1(b)). Figure 6.1(c) illustrates how a direction-dependent ranking can be used to avoid clustering in the target graph. Here, the output of the ranking method $\text{RANK}(x, \{y_1, \ldots, y_j\})$ is defined as follows. We first construct a sorted ring out of the set of input profiles $y_1, \ldots, y_j$ and the base node x. We then assign a rank value to each node defined as the minimal hop count to the node from x in this ring. The output of the ranking method is a list of the input profiles ordered according to this rank value. In this manner, the first 2α positions in the ranking contain α nodes preceeding x and α nodes following x in the sorted ring; hence the name “direction-dependent”.

Algorithm 13 T-MAN

```
1: loop
2:    wait(Δ)
3:    p ← selectPeer(ψ, rank(myDescriptor,view))
4:    sendPush(p, toSend(p, m))
5:
6: procedure ONPUSH(msg)
7:    sendPull(msg.sender, toSend(msg.sender,m))
8:    onPull(msg)
9: procedure ONPULL(msg)
10:   view.merge(msg.buffer)
11:
12: procedure TOSEND(p, m)
13:   buffer ← (MyDescriptor)
14:   buffer.append(view)
15:   buffer ← rank(p,buffer)
16:   return buffer.subList(1, m)
```

6.5 The T-MAN Protocol

As mentioned earlier, the T-MAN protocol is based on a gossiping scheme, in which all nodes periodically exchange node descriptors with peer nodes, thereby constantly improving the set of nodes they know—their partial views. Each node executes Algorithm 13. Notice that the algorithm fits in the scheme of the gossip protocols discussed so far. Any given view contains the descriptors of a set of nodes. As before, the view is a list data structure and it is also a set (each node has at most one descriptor on the view). Method MERGE is a set operation in the sense that it keeps at most one descriptor for each node. Parameter m denotes the message size as measured in the number of node descriptors that the message can hold. Method SELECTPEER selects a random sample among the first ψ entries in the list given as its second parameter.

In this section we do not specify how node views are initialized. In the rest of the chapter, we always describe the particular node view initialization procedure that we assume. These procedures include random initialization for the purposes of theoretical analysis in Section 6.6 and practical solutions based on various broadcasting schemes and realistic random peer sampling in Section 6.7.

We note that the protocol does not place a limit on the view size. This is done in order to decrease the number of parameters, thereby simplifying the presentation. One might expect that lack of a limit on view size might present scalability problems due to views growing too large. As we will show in Section 6.6, however, the storage complexity of nodes due to views grows only logarithmically as a function of the network size. Furthermore, preliminary experiments for the applications we consider show that imposing a comfortable limit on view sizes (larger than both m and K) does not result in any observable decrease in performance. This suggests that the simplification of ignoring view size limits is justified and is not critical for these applications. As usual, we define a cycle to be an interval of Δ time units.

Figure 6.2 illustrates the results of T-MAN for constructing a torus (visualizations were obtained using [136]). For this example, it is clear that only a few cycles are sufficient for convergence, and the target graph is already evident even after the first few cycles. In the next sections we will show that this rapid convergence is not unique to the torus example but that T-MAN performs well in a wide range of settings and that it is scalable, very similarly to epidemic broadcast protocols.

In Table 6.1 we summarize the parameters of the protocol. Note that K (target view size) controls the size of the target graph, and consequently, affects the running time of the protocol. For example, if we increase K while keeping the ranking method fixed, then the protocol will take longer to converge since it has to find a larger number of links.

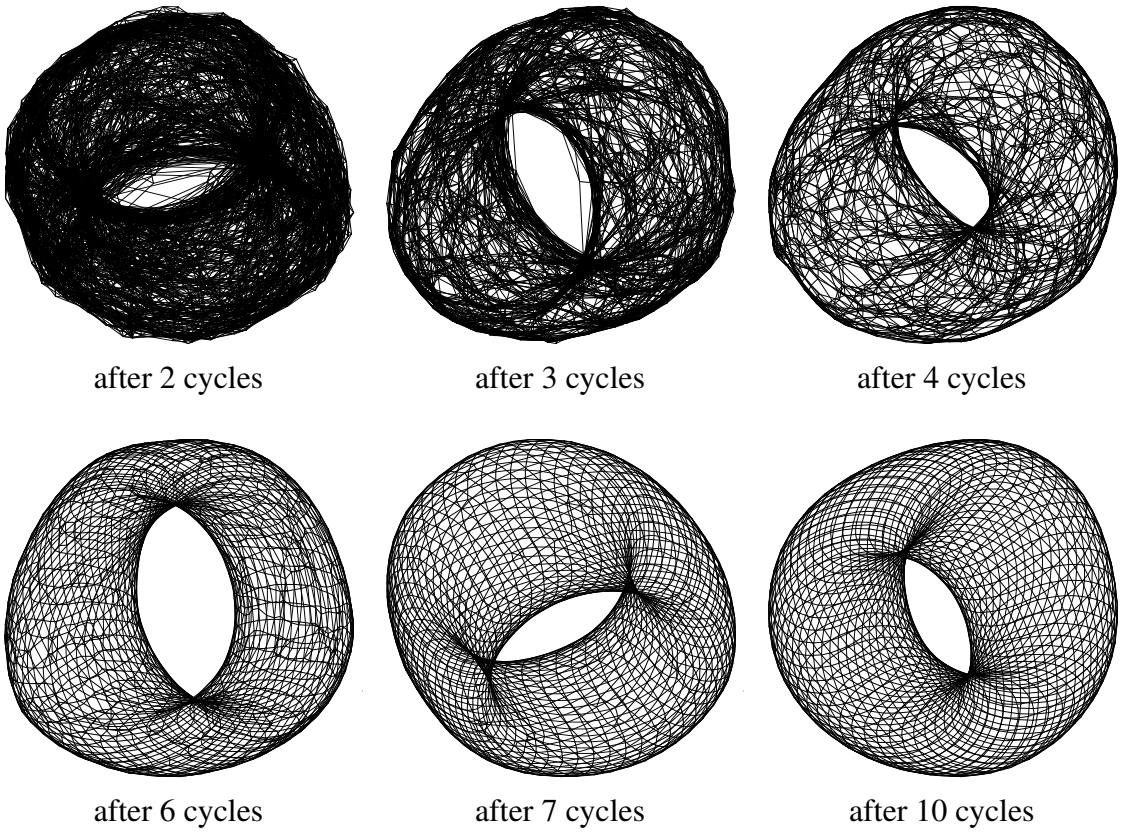


Figure 6.2: Illustration of constructing a torus over $50 \times 50 = 2500$ nodes, starting from a uniform random graph with initial views containing 20 random entries and the parameter values $m = 20, \psi = 10, K = 4$.

6.6 Key Properties of the Protocol

In this section we study the behavior of our protocol as a function of its parameters, in particular, m (message size), ψ (peer sampling parameter) and the ranking method RANK. Based on our findings, we will extend the basic version of the peer selection algorithm with a simple "tabu-list" technique as described below. Furthermore, we analyze the storage complexity of the protocol and conclude that on the average, nodes need $O(\log N)$ storage space where N is the network size.

To be able to conduct controlled experiments with T-MAN on different ranking methods, we first select a graph instead of a ranking method, and subsequently "reverse-engineer" an appropriate ranking method from this graph by defining the ranking to be the ordering consistent with the *minimal path length* from the base node in the selected graph. We will call this selected graph the *ranking graph*, to emphasize its direct relationship with the ranking method.

Note that the target graph is defined by parameter K, so the target graph is identical to the ranking graph only if the ranking graph is K-regular. However, for convenience, in this section we will not rely on K because we either focus on the dynamics of convergence (as opposed to convergence time), which is independent of K, or we study the discovery of neighbors in the ranking graph directly.

In order to focus on the effects of parameters, in this section we assume a greatly simplified system model where the protocol is initiated at the same time at all nodes, where there are no failures, and where messages are delivered instantly. While these assump-

RANK()	*Ranking method*: determines the preference of nodes as neighbors of a base node
K	*Target view size*: along with RANK(), it determines the target graph
Δ	*Cycle length*: sets the speed of convergence but also the communication cost
ψ	*Peer sampling parameter*: peers are selected from the ψ most preferred known neighbors
m	*Message size*: maximum number of node descriptors that can be sent in a single message

Table 6.1: Parameters of the T-MAN protocol.

tions are clearly unrealistic, in Section 6.7 we show through event-based simulations that the protocol is extremely robust to failures, asynchrony and message delays even in more realistic settings.

6.6.1 Analogy with the Anti-Entropy Epidemic Protocol

In Section 6.4 we defined the overlay construction problem with the help of a naive approach that involved the full dissemination of all the node descriptors to every node. Here we would like to elaborate on this idea further. Indeed, the anti-entropy epidemic protocol—when used to implement the naive approach—can be seen as a special case of T-MAN, where the message size m is unlimited (i.e., $m \geq N$ such that every possible node descriptor can be sent in a single message) and peer selection is uniform random from the entire network. In this case, independently of the ranking method, all node descriptors that are present in the initial views will be disseminated to all nodes. Furthermore, it is known that full convergence is reached in less than logarithmic time in expectation (see Chapter 1).

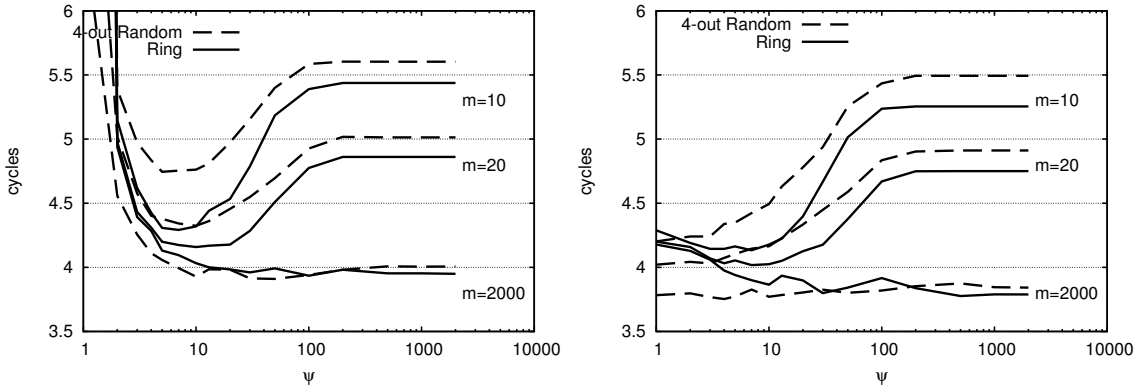
For this reason, the anti-entropy epidemic protocol is important also as a base case protocol when evaluating the performance of T-MAN, where the goal is to achieve similar convergence speed to anti-entropy, but with the constraint that communication is limited to exchanging a constant amount of information in each round. Due to the communication constraint, the performance will no longer be independent of the ranking method.

6.6.2 Parameter Setting for Symmetric Target Graphs

We define a symmetric target graph to be one where all nodes are interchangeable. In other words, all nodes have identical roles from a topological point of view. Such graphs are very common in the literature of overlay networks. The behavior of T-MAN is more easily understood on symmetric graphs, because focusing on a typical (average) node gives a good characterization of the entire system.

We will focus on two ranking graphs, both undirected: the ring and a k-out random graph, where k random out-links are assigned to all nodes and subsequently the directionality of the links is dropped. We choose these two graphs to study two extreme cases for the network diameter. The diameter (longest minimal path) of the ring is $O(N)$ while that of the random graph is $O(\log N)$ with high probability.

Let us examine the differences between realistic parameter settings and the anti-entropy epidemic dissemination scenario described above. First, assume that the message



(a) Basic T-Man protocol

(b) T-Man with Tabu List

Figure 6.3: Time to collect 50% of the neighbors at distance one in the ranking graph. Network size is $N = 2000$. Node views are initialized to contain 5 random links each. Graph (b) was obtained using a tabu list of size 4.

size m is a small constant rather than being unlimited. In this case, the random peer selection algorithm is no longer appropriate: if a node i contacts peer j that ranks low with i as the base node, then i cannot expect to learn new useful links from j because now (due to the small m) node j has a strong bias in its view towards nodes that rank high with j as a base node.

On the other hand, if a node i selects peers that rank too high with i as the base node, then convergence might slow down as well. The reason for this is that consecutive peers returned by the peer selection method will more often get repeated; in part because a node i is more likely to select a peer to communicate with that selected i shortly before, and in part because there are simply fewer nodes that are "close" to any given node than nodes that are far from it. This in turn results in increased correlation between the partial views of communicating partners, so the epidemic process is not maximally efficient.

Figure 6.3 illustrates this tradeoff using two ranking graphs: the ring and a random graph. The latter is generated by first constructing a 2-out directed regular random graph by selecting two random out-edges for each node, and subsequently taking the undirected version of this graph. The average degree of a node is thus 4, with a small variance. The basic version in Figure 6.3(a) applies the peer selection algorithm which picks a random peer from the highest ranking ψ nodes from the view, as described earlier. The point $\psi = N$ and $m = N$ corresponds to an anti-entropy epidemic dissemination (i.e., peer selection is unbiased and there are no limits on message size) which is optimal.

As predicted, with no limits on the message size ($m = N$), we can observe the effect due to the lack of randomness if the selected peer ranks too high (ψ is small). Furthermore, for large ψ performance again degrades when we place a limit on the message size since the correlation between communicating peers' ranking of the same set of nodes is reduced. This effect is less pronounced for larger m because now we might obtain useful information by chance even if there is little correlation between the rankings.

To verify our explanation as to why performance degrades with decreasing ψ, we apply a *tabu list* at all nodes in order to avoid contacting the same peers over and over again. The tabu list contains a fixed number of peers that a given node communicated with most recently. The node then does not initiate connection with any nodes in its

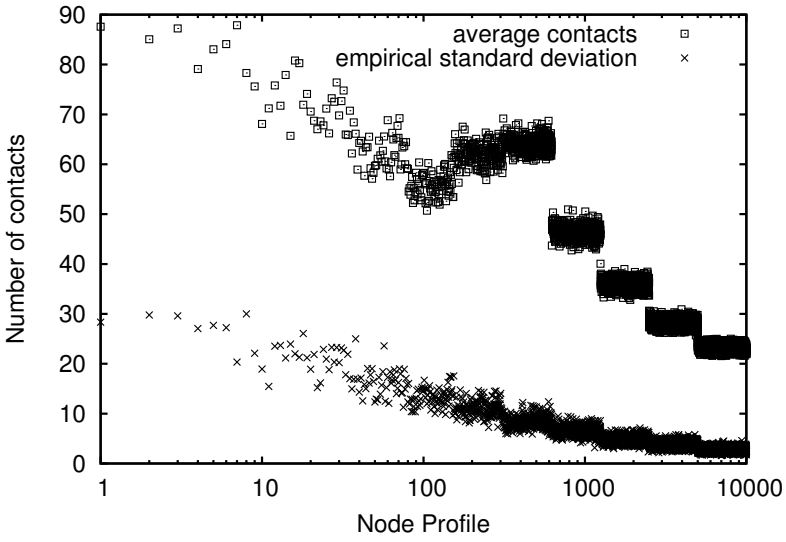


Figure 6.4: Number of contacts made by nodes while constructing a binary tree. Statistics are over 30 independent runs. The parameters are $N = 10000$, $m = 20$, number of cycles is 15, $\psi = 10$ and the tabu list size is 4. In the ranking graph, the root is node 0 and the out-links of node i are $2i + 1$ and $2i + 2$.

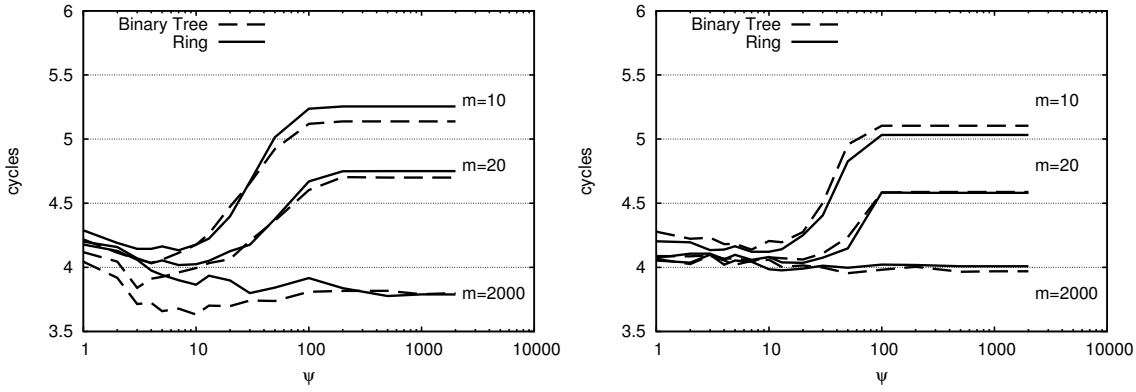
tabu list. We experimented with a tabu list size of 4. This mechanism does not add any communication overhead since it simply records the last 4 communications, but it is rather effective in reducing the negative effects of small ψ values as Figure 6.3(b) illustrates.

We can draw several other conclusions from the results in Figure 6.3. First, the tabu list slightly improves even the performance of anti-entropy epidemic dissemination with completely random peer selection ($m = \psi = N$). This is due to the fact that initially views contain only few nodes (to be precise, five, in this case). Without a tabu list, this significantly increases the chance of contacting the same peers in the first few cycles, while the views are still small. Such communications are not effective in advancing dissemination due to the correlated views of the communicating peers. Also note that when there is no limit on message size, the random graph outperforms the ring, especially when the tabu list is applied. This is due to the fact that the number of neighbors of a node in the random graph increases exponentially, so even for a small set of closest nodes, diversity is very high.

Finally, we note that the exponentially increasing neighborhood becomes a disadvantage when ψ is larger, because the view of peers that are further away from the base node in the ranking graph will be more uncorrelated to the view of the original peer. This suggests that for such graphs, peer selection should be aggressive ($\psi = 1$) and should be combined with the use of tabu lists.

6.6.3 Notes on Asymmetric Target Graphs

The topological role of nodes in asymmetric target graphs is not identical. For example, some nodes can be more central or more connected than others, there can be bridge nodes connecting isolated clusters, and so on. While symmetric graphs already exhibit complex behavior, we argue that asymmetric graphs cannot be treated reasonably in a common framework. Each case needs a separate analysis that needs to take into account the particular structure of the graph.



(a) T-Man with Tabu List

(b) T-Man with Tabu List and Balancing

Figure 6.5: Time to collect 50% of the neighbors at distance one in the ranking graph. The network size is $N = 2000$. Node views are initialized by 5 random links each. The tabu list size is 4.

To understand the problem better, consider a ranking method that is independent of the base node. This ranking method will induce a star-like structure since all nodes will be attracted to the very same high ranking nodes. In this case, more and more nodes will contact the nodes that rank high in the (in this case, common) ranking. As a result, convergence speeds up enormously, at the cost of a higher load on the central nodes. The reason is simple: the central nodes can collect the high ranking descriptors faster because they are contacted by many nodes. Due to their central position, they also distribute them very rapidly. One can even exploit this effect. For example, if the goal is to build a super-peer topology, with the high bandwidth nodes in the center, then the central nodes might actually be able to deal with the extra load, thus resulting in an efficient, but still fully self-organizing solution.

This effect can be observed in other interesting topologies as well. For example, rooted regular trees, where the non-leaf nodes have k out-links and one in-link, except the root, that has no in-links. If the ranking graph has such a topology, the resulting target graph will be asymmetric with highly nonuniform average traffic at nodes, as shown in Figure 6.4. One reason for this result is that a large proportion of the nodes are leaves. Leaf nodes, having only one neighbor, will have a tendency to talk to nodes that are further up in the hierarchy. This adds extra load on internal nodes and puts them in a more central position.

This in turn has a non-trivial effect on the convergence of the protocol, and allows T-MAN to have better performance for trees than for symmetric graphs. Figure 6.5 illustrates this effect. In Figure 6.5(a), we can observe the performance of T-MAN for a rooted and balanced binary tree as a ranking graph. We can see that there is a peculiar minimum when message size is unlimited but ψ is small. In this region, the binary tree consistently outperforms the ring, even for a small m.

This effect is due to the asymmetry of a binary tree. To show this, we ran T-MAN with an additional balancing technique, to cancel out the effect of central nodes. In this technique, we limit the number of times any node can communicate (actively or passively) in each cycle to two. In addition, nodes also apply *hunting* [20], that is, when a node contacts a peer, and the peer refuses the connection due to having exceeded its quota, the

node immediately contacts another peer until the peer accepts connection, or the node runs out of potential contacts. The results are shown in Figure 6.5(b). In the region of practical settings of ψ and m, the advantage of the binary tree disappears, while the ring preserves the same performance.

More detailed analysis reveals that in the initial cycles, nodes that are close to the root play a bootstrap function and communicate more than the rest of the nodes. After that, as the overlay network is taking shape, nodes that are further down the hierarchy take over the management of their local region, and so on. This is a rather complex behavior, that is *emergent* (not planned), but nevertheless beneficial. This also suggests that if the target graph is not symmetric, then extra attention is needed when explaining the behavior of T-MAN.

6.6.4 Storage Complexity Analysis

We derive an approximation for the storage space that is needed for maintaining views by the nodes (recall that there is no hard limit enforced by the protocol). This approximation is based on a number of simplifying assumptions that convert the problem into a model of disseminating news items, where only the most interesting news items can spread due to limited message size. Subsequently, we present experimental validation of the approximation using T-MAN on different realistic target graphs.

The News Spreading Model

To derive the approximation, we assume that the ranking method is independent of the base node, that is, all nodes rank a given set of node descriptors the same way. The rational for this assumption is the following. One conclusion of previous sections was that the success of T-MAN crucially depends on the fact that whenever a node i selects a peer j using SELECTPEER, the ranking of the current neighbors of i with node j as a base node is similar to the ranking with node i as a base node, because this way nodes i and j can provide relevant node descriptors to each other with a higher probability. Assuming that the ranking does not depend on the base node means that any selected node j is guaranteed to produce an identical ranking to node i, which is the ideal case for T-MAN.

This assumption, however, introduces a side-effect: it implies that the target graph is a star-like structure, with the m highest ranking nodes forming a clique, and all the other nodes pointing to these m nodes. This level of asymmetry is highly non-typical and therefore is an unrealistic scenario for T-MAN. To "fix" this side-effect, we assume that SELECTPEER returns a random node from the entire network, which makes the role of all nodes identical.

In this setting, node descriptors have no relation to actual nodes anymore (that is, the node addresses in the descriptors are never used), so we can think of the model as spreading *news items* that have a natural ranking based on "interestingness".

In the following we present a simplified deterministic model of the dynamics of news spreading to obtain a heuristic baseline, to which the storage complexity of T-MAN can be compared. Let $n_{i,j}(t)$ denote the number of news items of rank j at node i at time t. The value of $n_{i,j}(t)$ is 0 or 1, and $n_{i,j}(t)$ is monotonically increasing, because we assumed that the local view size is not bounded.

First of all, if $m \geq N$ then the values $n_{i,j}(t)$ $(i, j \in \{1, \ldots, N\})$ are identically distributed random variables, since there is no competition between the news items for the

slots available for spreading. Clearly, when $m < N$, then $E(n_{i,j}(t))$ can grow undisturbed until the effect of the competition kicks in, in other words, until the item with rank j is no longer competitive for the available m slots. Motivated by this, in our simplified deterministic model we will approximate $n_{i,j}(t)$ by

$$\hat{n}_{i,j}(t) = \begin{cases} E(n_{i,j}(t) \mid m \geq N) & \text{if } t \leq t^* \\ E(n_{i,j}(t^*) \mid m \geq N) & \text{if } t > t^* \end{cases} \tag{6.1}$$

where t^* is the point in time when $\hat{n}_{i,j}$ stops growing due to the fact that there are already enough more interesting news items at the local node i so that the item with rank j will no longer be included in the most interesting m items.

For $j \leq m$ we know that $t^* = \infty$, because these items will always be included in any message sent. For $j > m$, this point in time can be calculated using the fact that at time t^*

$$\sum_{k=1}^{j} \hat{n}_{i,k}(t^*) = j\hat{n}_{i,j}(t^*) = m, \tag{6.2}$$

where the first equation comes from the fact that $n_{i,j}(t)$ $(i, j \in \{1, \ldots, N\})$ are identically distributed. (Note that although t^* can be calculated, we do not actually need to calculate it, we simply need to know only that it is well defined.) This means that we have

$$\hat{n}_{i,j}(t) = \frac{m}{j}, \quad \text{for } t > t^*, j > m. \tag{6.3}$$

Figure 6.6 compares the prediction of this model and the converged distribution obtained experimentally via simulation with T-MAN. The figure uses the notation $n_j = \sum_i n_{i,j}$, which is the overall number of news items of rank j in the network. The indicated prediction is, accordingly, $\sum_i \hat{n}_{i,j} = Nm/j$.

Equation (6.3) allows us to approximate the actual storage space that is required for the views of the nodes. We focus only on the items that rank lower than m, because the highest ranking m items will be stored by all the nodes taking a constant amount of space. The sum of all entries with a rank higher than m stored in the system after convergence (when all messages are composed of the m most popular items already) is

$$\sum_{j=m}^{N} \hat{n}_j = \sum_{j=m}^{N} \frac{Nm}{j} = Nm \sum_{j=m}^{N} \frac{1}{j} = \Theta(N \log N), \tag{6.4}$$

where we used the well known approximation of the harmonic number and the fact that m is constant. Therefore each view stores $\Theta(\log N)$ entries on the average. Note that this result is independent of the number of iterations executed to reach convergence, and it is also independent of the actual form of the function $\hat{n}_{i,j}(t)$; recall that the only assumption we made was that these functions are monotonically increasing.

Finally, we note that $\hat{n}_j = Nm/j = Nmj^{-1}$ results in a power law distribution, as it follows the form $j^{-\gamma}$. Power laws are very frequently observed in complex evolving networks [114]. The phenomenon is often due to some form of "the rich get richer" effect. One can link our results to the study of other complex networks, for example, social networks. All nodes start with a random constant-size set of news items, and they gossip always only the m most interesting ones that they currently know. This dynamics results in a power law distribution of news items, with the most interesting news being known to everyone. Furthermore, each participant learns only about $\Theta(\log N)$ news items from the overall N news items available.

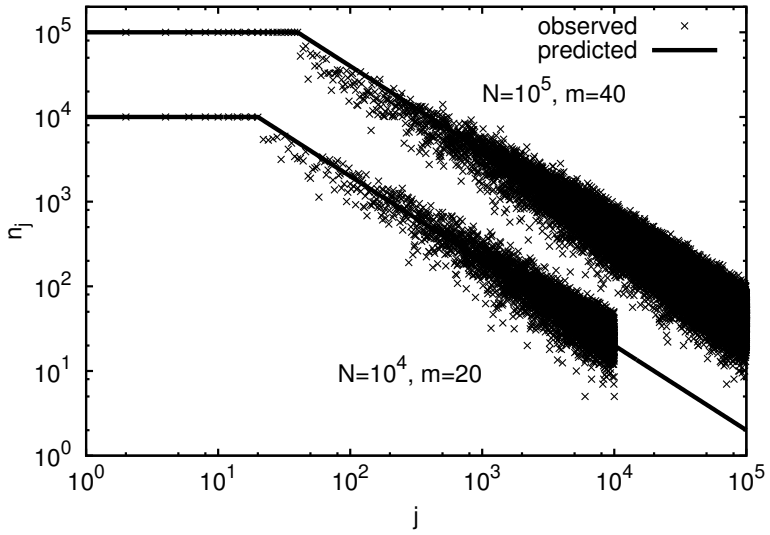


Figure 6.6: Experimental results and values predicted by Equation (6.3) for n_j with two sets of parameters $N = 10^4, m = 20$ and $N = 10^5, m = 40$. For each j, the converged value of n_j is indicated as a separate point. The observed values correspond exactly to the predicted one for the initial constant section, and are covered by the line segment on the graph.

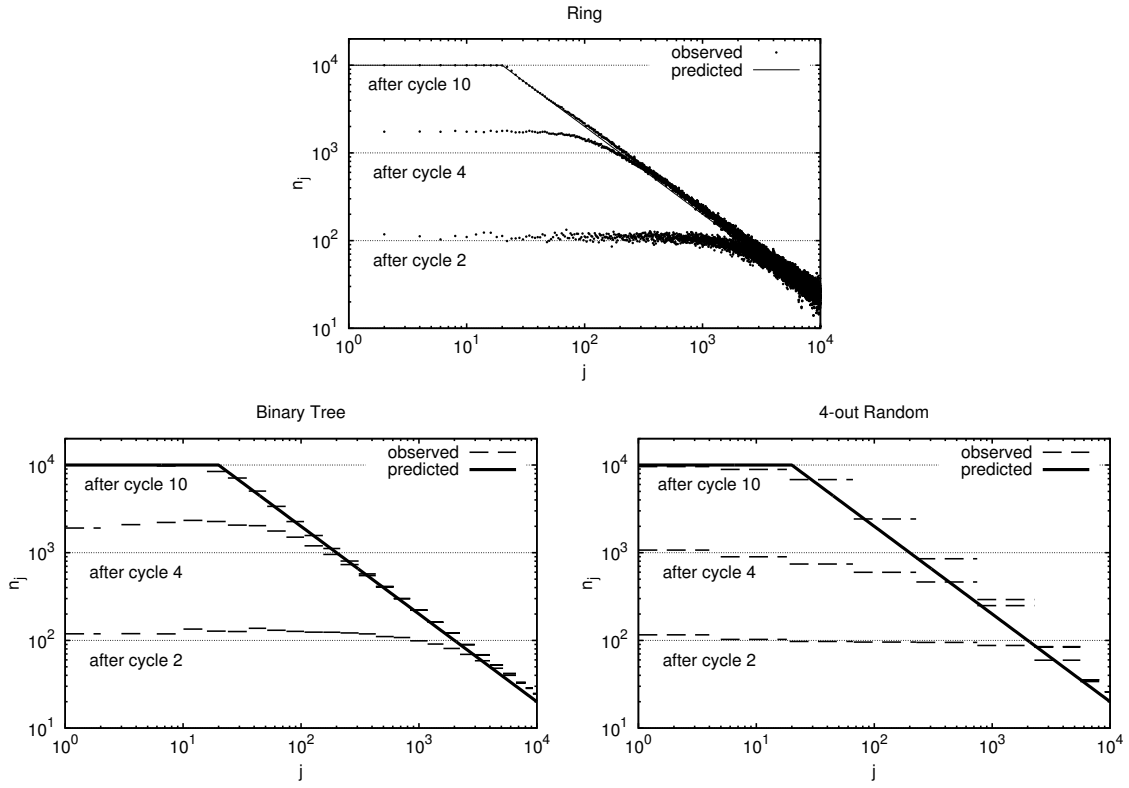


Figure 6.7: Experimental and predicted values of n_j for three different ranking graphs. Experiments were run with $N = 10^4$, $m = 20$ and $\psi = 10$, without a tabu list. Note that the plots contain three snapshots of the simulation for cycles 2, 4 and 10.

Empirical Validation

We verify experimentally that the prediction in (6.3) holds for T-MAN when different ranking methods are employed. This would support as a consequence the claim that Equation (6.4) characterizes the storage complexity of the protocol.

We need to generalize n_j since ranking can now depend on the base node. Let n_j be the number of nodes that know about the node with rank j according to their *own* ranking of the entire network. Figure 6.7 shows the values of n_j for three ranking graphs at three different times. Although the experiments reported in Figure 6.7 were performed without a tabu list, further experiments (not shown) show that tabu lists have no observable effect on the distribution of ranks in the views. They only speed up convergence of the protocol as discussed earlier.

In Figure 6.7 we can observe that the ring fulfills the assumptions of Section 6.6.4 best: the n_j values that have not stopped growing have the same value at each time point, which means they indeed grow at the same rate. The largest deviation can be observed in the case of the random graph. There, the growth of the n_j values slows down smoothly which implies that the assumption they grow at the same rate does not hold. This results in a slight "overshoot" where the observed values are slightly higher than those predicted.

Note that in the case of the binary tree, the predicted values match closely the observed ones even though the topology is not symmetric. This further underlines the robustness of the prediction. In other words, the seemingly strong assumptions of the theory in fact leave the essential dynamics almost unchanged, which indicates that we could understand important features of the protocol. Of course, the more central nodes need more storage capacity, the prediction holds only on average. However, in our preliminary experiments (not shown), we have seen that setting a reasonable hard limit on the view size that is significantly larger than m (for example, 1000 items) does not result in any significant difference in performance. For this reason we opted for the simplified discussion and we omit hard limits on the view size in the following.

6.7 Experimental Results

In the previous section we considered the most basic version of the protocol to shed light on its convergence properties and storage complexity. This section is concerned with developing additional techniques that allow for the practical application of the protocol; in particular, we address two important problems: how to start and how to stop the protocol. We also present an extensive empirical analysis under different parameter settings and different failure scenarios, introduced by a brief discussion of the simulation environment and the figures of merit we focus on.

6.7.1 A Practical Implementation

So far we assumed that the protocol is started at all nodes at once, in a synchronous fashion, and we were not dealing with termination at all. We also assumed that at all nodes the initial set of known peers is a random sample from the network. In this section, we replace these unrealistic assumptions with practically feasible solutions.

Peer Sampling Service

The peer sampling service provides each node with continuously up-to-date random samples of the entire population of nodes. Such samples fulfill two purposes: they enable the random initialization of the T-MAN view, as discussed in Section 6.5, and make it possible to implement a starting service as well, allowing for the deployment of various gossip based broadcast and multicast protocols.

We consider an instantiation of the peer sampling service based on the NEWSCAST protocol (see Section 2.2.4), chosen for its low cost, extreme robustness and minimal assumptions. The basic idea of NEWSCAST is that each node maintains a local set of random node addresses: the (partial) *view*. Periodically, each node sends its view to a random member of the view itself. When receiving such a message, a node keeps a fixed number of freshest addresses (based on timestamps), selected from those locally available in the view and those contained in the message.

Each node sends one message to one other node during a fixed time interval. Implementations exist in which these messages are small UDP messages containing approximately 20-30 IP addresses, along with the ports, timestamps, and descriptors such as node IDs. The time interval is typically long, in the range of 10 s. The cost is therefore small, similar to that of heartbeat messages in many distributed architectures. The protocol provides high quality (i.e., sufficiently random) samples not only during normal operation (with relatively low churn), but also during massive churn and even after catastrophic failures (up to 70% nodes may fail), quickly removing failed nodes from the local views of correct nodes.

Starting and Terminating the Protocol

We implemented a simple starting mechanism based on well-known broadcast protocols. The content of the broadcast message may be a simple "wake up" specifying *when* to build a predefined network, or it may include additional information specifying *what* network to build (e.g., by providing the implementation of a specific ranking function). To simplify our simulation environment, we adopt the first approach; technical issues related to the second one may be easily solved in a real implementation.

The following terminology is used when discussing the starting mechanism. We say that a node is *active* if it is aware of and explicitly participating in a specific instance of T-MAN; if the node is not aware that a protocol is being executed, it is called *inactive*.

Initially, there is only one active node, the *initiator*, activated by an external event (e.g., a user's request). An inactive node may become active by exchanging information with nodes that are already active. When a node becomes active, it immediately starts executing the T-MAN protocol. The final goal is to activate all nodes in the system, i.e., to start the protocol at all nodes.

The actual implementation of the broadcast can take many forms that differ mainly in communication overhead and speed.

Flooding As soon as a node becomes active for the first time, it sends a "wake up" message to a small set of random nodes, obtained from the peer sampling service. Subsequently, it remains silent.

Anti-Entropy, Push-only Periodically, each active node selects a random peer and sends a "wake-up" message [20].

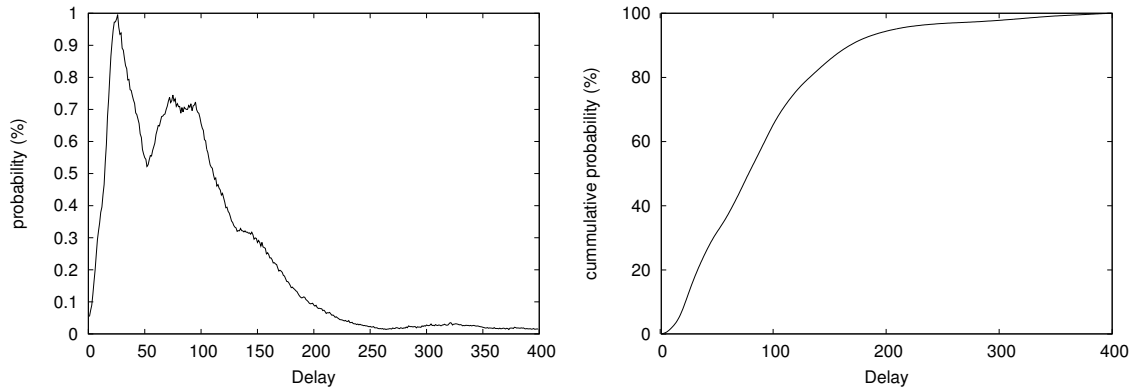


Figure 6.8: Probability distribution of end-to-end delays as reported in the King data set [137].

Anti-Entropy, Push-Pull Periodically, each node (active or not) exchanges its activation state with a random peer. If either of them was active, they both become active [20].

As described above, a node becomes active as soon as it receives a message from another active node. Note, however, that messages belonging to the starting protocol are not the only source of activation; a node may also receive a T-MAN message, from a node that has already started to execute the protocol. This message also activates the recipient node.

As is well known, flooding is fast and effective but very expensive due to message duplications. In comparison, the most important advantage of the other two approaches is the dramatically lower communication overhead per unit time. The overhead can further be reduced to almost zero, due to the fact that the starting service messages can be piggybacked, for example, on NEWSCAST messages that implement the peer sampling service.

After the target graph has been built, the protocol does not need to run anymore and therefore must be terminated. Clearly, detecting global convergence is difficult and expensive: what we need is a simple local mechanism that can terminate the protocol at all nodes independently.

We propose the following mechanism. Each node monitors its own local view. If no changes (i.e., node additions) are observed for a specified period of time (δ_{idle}), it suspends its active thread. We call this state *suspended*. If a view change occurs when a node is suspended (due to an incoming message initiated by another node that is still active), the node switches again to the active state, and resets its timer that measures idle time.

6.7.2 Simulation Environment

All the experiments are event-based simulations, performed using PEERSIM, an open-source simulator designed for large-scale P2P systems and publicly available at SourceForge [66]. The applied transport layer emulates end-to-end delays between pairs of nodes based on the traces of the King data set [137]. Delays reported in these traces range from 1 ms to 400 ms, and the probability distribution is as shown in Figure 6.8.

The following parameters are fixed in the experiments: the size of the tabu list is 4, and the peer selection parameter (ψ) is 1. If different values are not explicitly mentioned, the message size (m) is 20, the cycle length (Δ) is 1 s, and the value of δ_{idle} is set to 4 s. Each

experiment is repeated 50 times with different random seeds. Plots show the average of the observed measures, along with error bars; when graphically feasible, individual experiments are displayed as separate dots with a small random translation.

6.7.3 Ranking Methods

To emphasize the robustness of T-MAN to the actual target graph being built, we performed all experiments on two different tasks: building a sorted ring, and building a binary tree. These two graphs have very different topologies: the ring has a large (linear) diameter while the tree has a small (logarithmic) one. Besides, as pointed out in Section 6.6.3, in the tree some nodes are more central than others, while in the ring all nodes are equal from this point of view.

In the previous sections, we applied the concept of a ranking graph to (implicitly) define the ranking method. This approach is not practical, so we need to define explicit and locally computable ranking methods.

Sorted Ring

Creating a sorted ring is very useful, for example, for the decentralized computation of the ranking of nodes [13] or jump-starting distributed hash tables, such as CHORD [83]. The latter application is further discussed in Chapter 7.

We assume that the node profile is an element of a collection, over which a total ordering relation is defined. In particular, we work with 60-bit integers as node profiles that are initialized at random for each node. We want the target graph to be a ring, in which the node profiles are ordered (except one pair where the largest and smallest values meet) to close the ring.

To achieve this target graph, the output of the ranking method $\text{RANK}(x, y_1, \ldots, y_k)$ is defined as follows. First we construct a sorted ring (as defined above) out of the set of input profiles $y_1, \ldots, y_k$ and the base node x, and assign a rank value to all nodes: the minimal hop count from x in this ring. The output of the ranking method is an ordered list of the input profiles according to these assigned rank values. Note that this is a *direction-dependent* ranking method, that cannot be induced by a distance metric over the node profiles. For simplicity, we will call T-MAN with this ranking method SORTED RING.

Binary Tree

The second topology we consider is an undirected rooted binary tree. To achieve a well controlled target graph for the sake of experimental comparison, the node profiles are defined as follows. If there are N nodes, then we assign the integers $1, \ldots, N$ to the nodes in some arbitrary order. The node with value 1 is the root. Using the binary representation of these integers, the node $0a_2 \ldots a_m$ has two children: $a_2 \ldots a_m 0$ and $a_2 \ldots a_m 1$. Numbers starting with 1 belong to leafs.

It is easy to calculate the shortest path length in this tree between two arbitrary nodes, based on the two node profiles. This notion of distance is used to define the ranking function required by T-MAN to build the tree: $\text{RANK}(x, y_1, \ldots, y_k)$ sorts the input profiles $y_1, \ldots, y_k$ according to distance from the base node x. For simplicity, we will call T-MAN with this ranking method TREE.

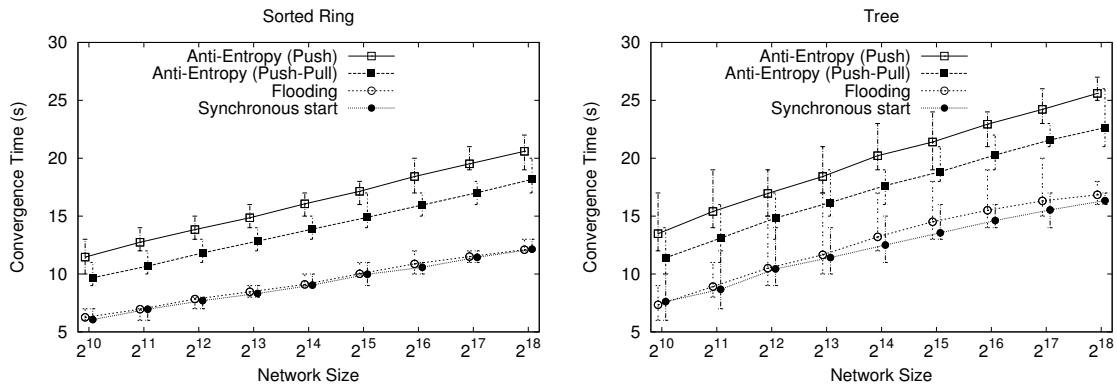


Figure 6.9: Convergence time as a function of size, using different starting protocols.

6.7.4 Performance Measures

We are interested both in the effectiveness (speed and quality) and efficiency (cost) of the protocol. We evaluate our protocols using the following performance measures: *convergence time*, *target links found*, *termination time* and *communication costs*.

convergence time The time needed to obtain the *perfect* target graph. In the case of SORTED RING, each node must know at least its first successor and predecessor in the sorted ring. For TREE, each node different from the root must know its parent, and non-leaf nodes must know their children.

target links found The number of links in the target graph that are actually found by T-MAN at a certain time, typically at termination time. This allows for a more fine-grained assessment of performance than convergence time.

termination time The total time needed to complete (start, execute and stop) the protocol at *all* nodes. This may be considerably longer than convergence time, although, as we will see, typically only few nodes are still active after reaching convergence.

communication cost The number of messages exchanged. Note that all messages ever exchanged are of the same size.

The unit of time will be cycles or seconds, depending on which is more convenient (note that cycle length defaults to 1 s). We also note that convergence time is not defined if the protocol terminates before converging. In this case, we use the number of identified target links as a measure.

6.7.5 Evaluating the Starting Mechanism

Figure 6.9 shows the convergence time for SORTED RING and TREE, using the starting protocols described in Section 6.7.1. The cycle length of the anti-entropy versions was the same as that of T-MAN, and the flooding protocol used 20 random neighbors at all nodes. The case of synchronous start is also shown for comparison. Note that these figures do not represent a direct measure of the performance of well-known starting protocols; rather, convergence time plotted here represents the overall time needed to both start the protocol and reach convergence, with T-MAN and the broadcast protocol running concurrently.

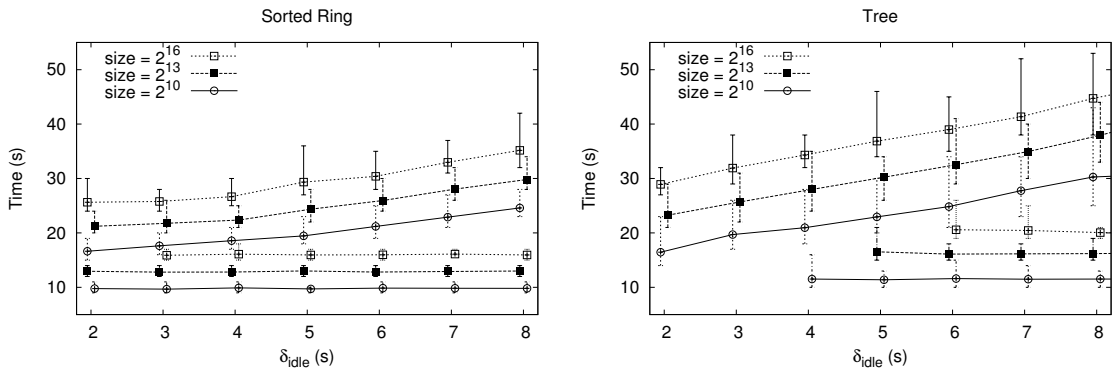


Figure 6.10: Convergence time (bottom curves) and termination time (top curves) as a function of δ_{idle}.

In the case of flooding, "wake-up" messages quickly reach all nodes and activate the protocol; almost no delay is observed compared to the synchronous case. Anti-entropy mechanisms result in a few seconds of delay. In the experiments that follow, we adopt the anti-entropy, push-pull approach, as it represents a good trade-off between communication costs and delay. Note however that (unlike the push approach) the push-pull approach assumes that at least the starting service was started at all nodes already.

6.7.6 Evaluating the Termination Mechanism

We experimented with various settings for δ_{idle} ranging from 2 s to 12 s. Figure 6.10 shows both convergence time (bottom three curves) and termination time (top three curves) for different values of δ_{idle}, for SORTED RING and TREE, respectively. In both cases, termination time increases linearly with δ_{idle}. This is because, assuming the protocol has converged, each additional cycle to wait simply adds to the termination time.

For small values convergence was not always reached, especially for TREE. For SORTED RING, all runs converged except the case when $\delta_{idle} = 2$ and $N = 2^{16}$, when 76% of the runs converged. For TREE, all runs converged with $\delta_{idle} > 5$ and no runs converged for $(\delta_{idle} = 2, N = 2^{13})$, $(\delta_{idle} = 2, N = 2^{16})$, and $(\delta_{idle} = 3, N = 2^{16})$. Even in these cases, the quality of the target graph at termination time was almost perfect, as shown in Figure 6.11. In the worst of our experiments, we observed that no more than 0.1% of the target links were missing at termination. This may be sufficient for most applications, especially considering that the target graphs will never be constructed perfectly in a dynamic scenario, where nodes are added and removed continuously. Nevertheless, from now on, we discard the parameter combinations that do not always converge.

Apart from longer executions, an additional consequence of choosing large values of δ_{idle} is a higher communication cost. However, since not all nodes are active during the execution, the overall number of messages sent per node on average is less than one quarter of the number of cycles until global termination. To understand this better, Figure 6.12 shows how many nodes are active during the construction of SORTED RING and TREE, respectively. The curves show both an exponential increase in the number of active nodes when starting, and an exponential decrease when stopping. The period of time in which all nodes are active is relatively short.

These considerations suggest the use of higher values for δ_{idle}, at the cost of a larger termination time and a larger number of exchanged messages. The chosen value of δ_{idle}

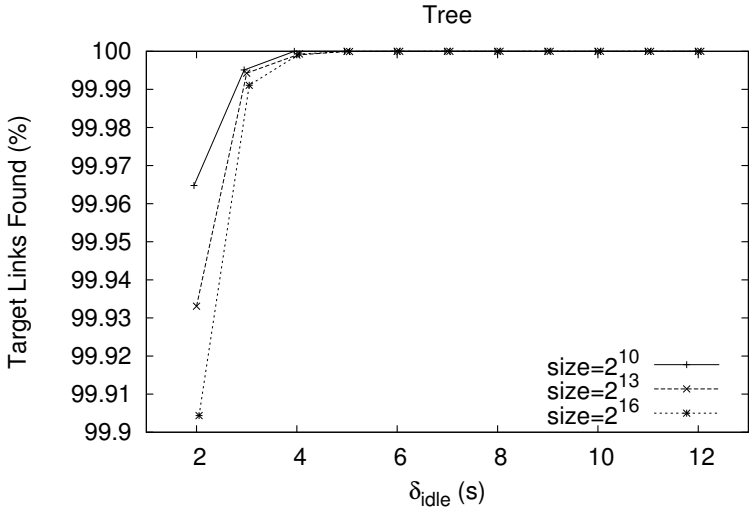


Figure 6.11: Quality of the target TREE graph at termination time as a function of δ_{idle}

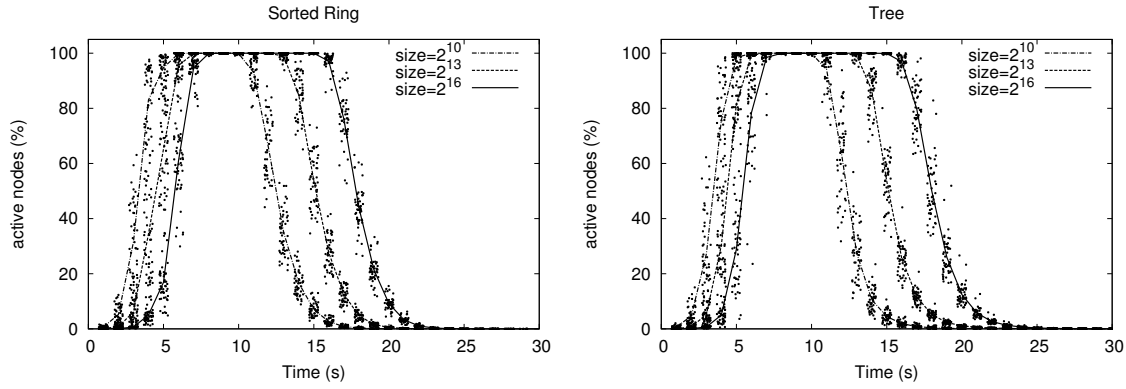


Figure 6.12: Proportion of active nodes during execution.

(4 s) represents a good trade-off between the desire of obtaining a perfect target graph and the consequently larger cost in time and communication.

6.7.7 Parameter Tuning

Cycle Length If a faster execution is desired, one can always decrease the cycle length. However, after some point, decreasing cycle length does not pay off because message delay becomes longer than the cycle length and eventually the network will be congested by T-MAN messages. Figure 6.13 shows the behavior of T-MAN with a cycle length varying between 0.2 s and 4 s. The figure shows the number of cycles required to terminate the protocol. Small cycle lengths require a larger number of cycles, while after a given threshold (around 1 s), the number of cycles required to complete a protocol is almost constant. The reason for this behavior is that with short cycles, multiple cycles may be executed before a message exchange is concluded, thus wasting bandwidth in sending and receiving old information multiple times.

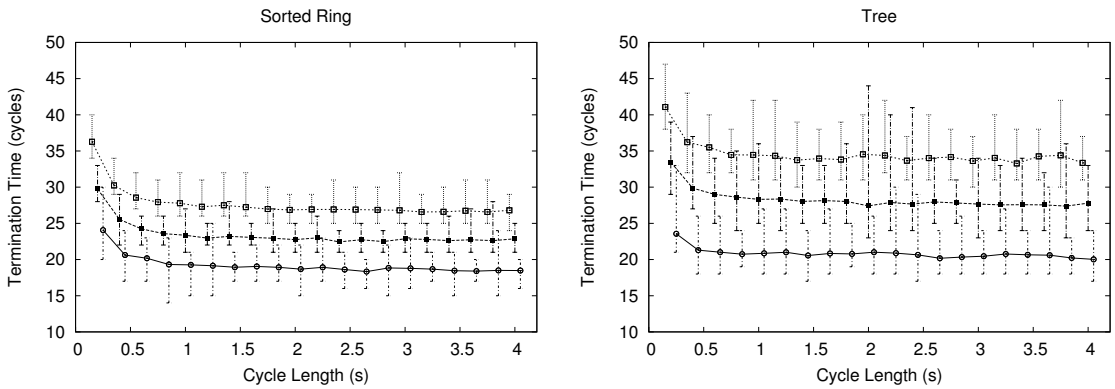


Figure 6.13: Termination time as a function of cycle length.

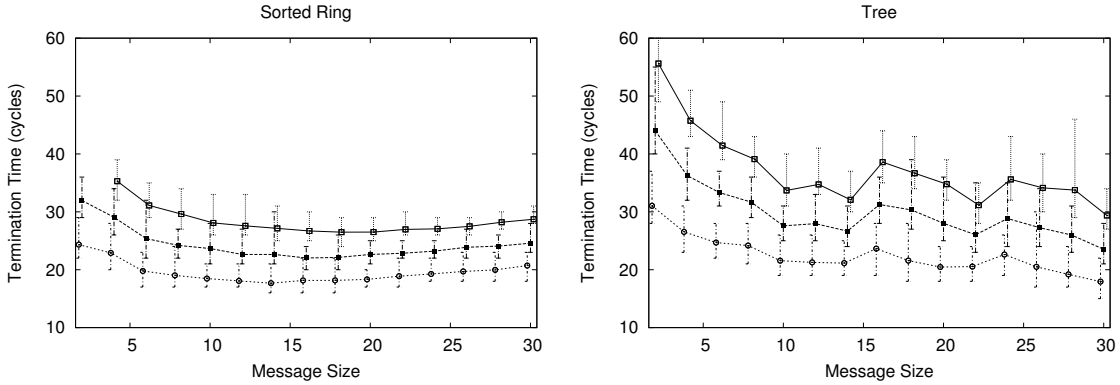


Figure 6.14: Termination time as a function of message size.

Message Size In Section 6.6, we have examined the effect of the message size parameter (m) in detail. Here we are interested in the effect of message size on termination time. Figure 6.14 shows that by increasing the size of messages exchanged by SORTED RING termination time slightly increases after around $m = 20$. The reason is that a node becomes suspended only after the local view remains unchanged for a fixed number of cycles, but increasing the message size has the effect of increasing the number of cycles in which view changes might occur, thus delaying termination. The results for TREE have more variance, which might have to do with the unbalanced nature of the topology, as discussed in Section 6.6.3.

6.7.8 Failures

The results discussed so far were obtained in static networks, without considering any form of failure. Here, we consider two sources of failure: message losses and node crashes. Since in this chapter we consider only the overlay *construction* problem, and not *maintenance*, we do not explicitly consider scenarios involving node churn. Instead, we model churn through nodes leaving, and do not allowing joining nodes to participate in an ongoing construction. Furthermore, since we do not have a leave protocol, leaving nodes are identical to crashing nodes from our point of view.

Message Loss While a simple solution could be to adopt a reliable, connection-oriented transport protocol like TCP, it is more attractive to rely on a lightweight but perhaps

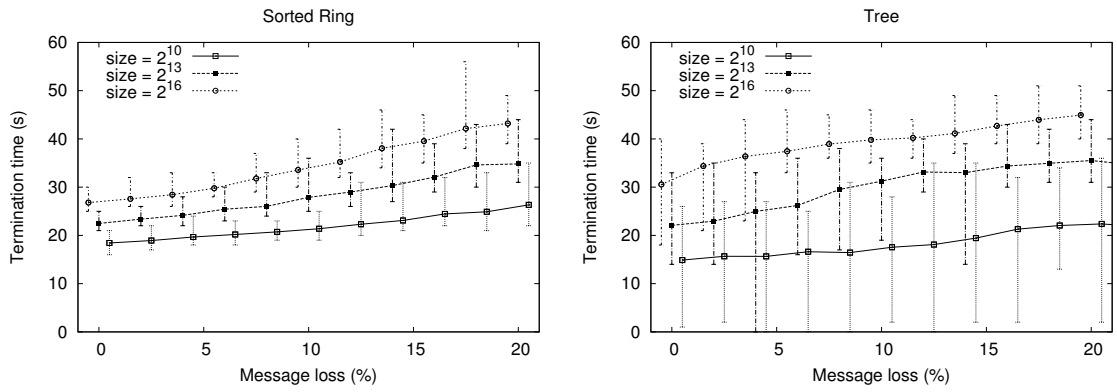


Figure 6.15: Termination time as a function of message loss rate.

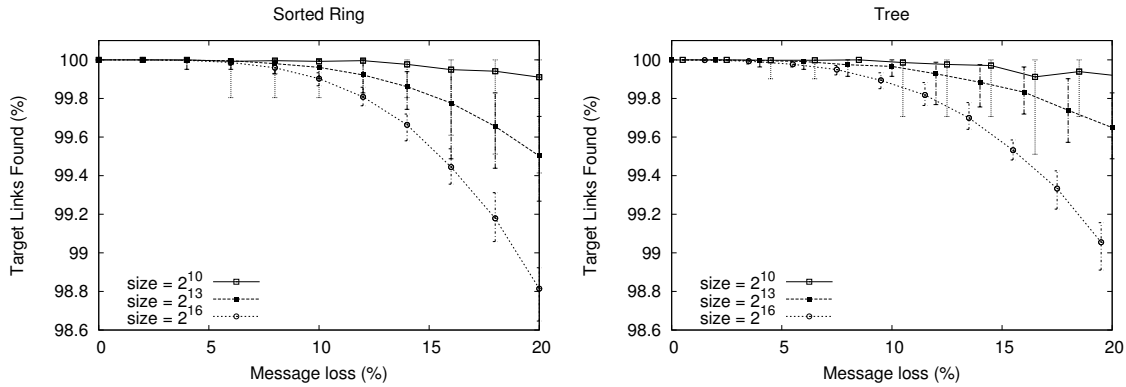


Figure 6.16: Target links found by the termination time as a function of message loss rate.

unreliable transport. In this case, we need to demonstrate that T-MAN can cope well with message loss. Figure 6.15 shows that T-MAN is highly resilient to message loss and so a datagram-oriented protocol like UDP is a perfectly suitable choice, as message losses only slow down the protocol slightly. Many message exchanges are either never started or never completed, thus requiring more cycles to terminate the protocol execution. The *quality* does not suffer much either. In both SORTED RING and TREE, around 1% of the target links may be missing, as shown by Figure 6.16. Note that the mean message loss ratio for geographic networks like the Internet is around 2% [138], an order of magnitude smaller than the maximum message loss ratio tested in our experiments.

Node Crashes Figure 6.17 shows the behavior of T-MAN with a variable failure rate, measured as the total number of nodes leaving the network per second per node. We experimented with values ranging from 0 to 10^{-2}, which is two orders of magnitude larger than the value of 10^{-4} suggested as the typical behavior of some P2P networks [139]. The results show that both SORTED RING and TREE are robust in normal scenarios, with TREE being considerably more reliable in the range of extreme failure rates. This is due to the unbalanced nature of the topology as discussed in Section 6.6.3.

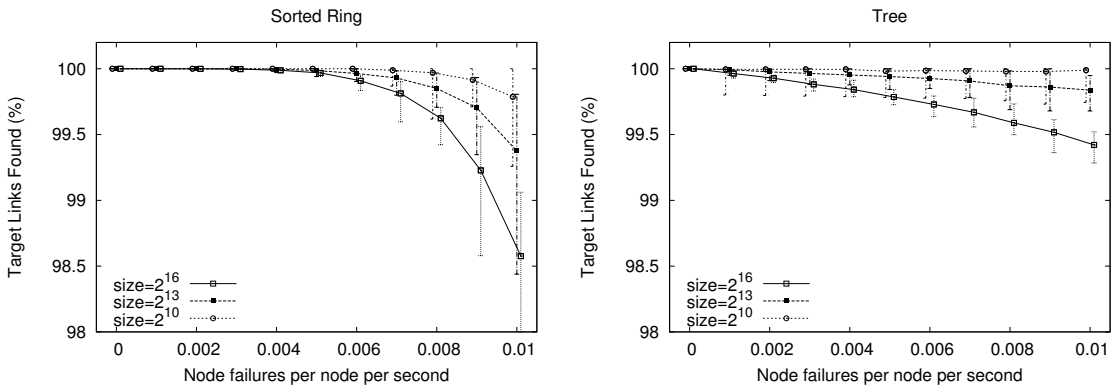


Figure 6.17: Target links found by the termination time as a function of failure rate.

6.8 Conclusions

We presented T-MAN, a lightweight gossip-based protocol for constructing various overlay networks. The target network is given by the ranking method, which is a parameter of the protocol. T-MAN is robust to the target network: it exhibits good performance that is mostly invariant over a wide range of target networks such as rings and trees. The protocol is simple and robust to failure scenarios which makes it attractive for practical applications.

In closing, we note that T-MAN has been successfully applied for constructing the CHORD overlay network (see Chapter 7) and the PASTRY overlay network (see Chapter 8). In this chapter, we have chosen to focus on overlay construction as opposed to overlay maintenance. Possible overlay maintenance techniques involve limited local view sizes and periodic removal of old entries from the view. In addition, random samples from the network can constantly be injected into the local view.

Chapter 7

Bootstrapping Chord

In this chapter we describe a practically relevant application of T-MAN: we use it to create a CHORD network [83]. Structured peer-to-peer overlay networks like CHORD are now an established paradigm for implementing a wide range of distributed services. While the problem of maintaining these networks in the presence of churn and other failures is the subject of intensive research, the problem of *building* them from scratch has not been addressed (apart from individual nodes joining an already functioning overlay). Here, we address the problem of *jump-starting* a popular structured overlay, CHORD, from scratch.

This problem is of crucial importance in scenarios where one is assigned a limited time interval in a distributed environment such as a data center for Cloud computing, and the overlay infrastructure needs to be set up from the ground up as quickly and efficiently as possible, or when a temporary overlay has to be generated to solve a specific task *on demand*.

We introduce T-CHORD, that can build a CHORD network efficiently starting from a random unstructured overlay. After jump-starting, the structured overlay can be handed over to the CHORD protocol for further maintenance. We demonstrate through extensive simulation experiments that the proposed protocol can create a perfect CHORD topology in a logarithmic number of steps. Furthermore, using a simple extension of the protocol, we can optimize the network from the point of view of message latency.

7.1 Introduction

Structured overlay networks have received considerable attention recently [81, 83]. A wide range of distributed services and applications can efficiently be implemented on top of structured overlays. The fundamental abstraction that is the basis of numerous applications is *key-based routing* [80]. Key-based routing protocols are based on *routing tables* stored at each node and that are used to forward messages for a specific key towards the destination: the node that is responsible for the given key. The neighborhood relations specified by the routing tables define the overlay topology, whose structure depends on the specific implementation.

While the problem of maintaining these networks in the presence of churn and other failures is the subject of intensive research, the problem of *building* them from scratch has not been addressed apart from handling node joins to an existing overlay. Yet, in some important scenarios, we face the problem of *jump-starting* structured overlays from scratch. This problem gains particular importance if one is assigned a limited time interval in a distributed environment such as PlanetLab [95], or a Grid [140], and the overlay

infrastructure needs to be set up from the ground up as quickly and efficiently as possible, or when a temporary overlay has to be generated to solve a specific task *on demand*.

Existing join protocols are not designed to handle the massive concurrency involved in a jump-starting process, when all the nodes are trying to join at the same time [83]. On the other hand, naive approaches where nodes are forced to join the overlay in some specified order results in at least linear time needed to construct the network (not to mention the serious problem of synchronizing the operations).

We propose a solution to the jump-starting problem called T-CHORD that is simple, scalable, robust, and efficient. T-CHORD is a protocol for bootstrapping the CHORD topology *on demand* starting from an unstructured, uniform random overlay. The purpose of T-CHORD is purely jump-starting the overlay; the constructed network is handed over to the CHORD protocol for further maintenance.

T-CHORD is based on T-MAN. As we have seen in Chapter 6, T-MAN is a generic mechanism for building and maintaining a wide range of different topologies, including rings, grids and trees. Briefly, T-CHORD works as follows. We assume the existence of a connected unstructured overlay network characterized by a random topology (such as those produced by protocols in Chapter 2). Nodes are assigned unique IDs from a circular ID space. Starting from the initial random overlay, T-MAN is used to build the ring to be used by CHORD for consistent routing. At all nodes, as a "side effect" of its execution (by remembering all the encountered nodes), T-MAN can also provide a larger set of nodes from which CHORD fingers can be selected.

We have evaluated the topologies obtained by T-CHORD through simulation. The results, presented in Section 7.4, confirm that the obtained topology is equivalent to (in fact, at times slightly better than) the "optimal" CHORD topology (as defined in the CHORD protocol specification) based on routing performance: loss rate, hop count and latency.

7.2 System Model

We consider a network consisting of a large collection of *nodes* that are assigned unique identifiers and that communicate through message exchanges. The network is highly dynamic; new nodes may join at any time, and existing nodes may leave, either voluntarily or by *crashing*. Since voluntary leaves can be trivially managed by simple "logout" protocols, in the following we limit our discussion to node crashes, that are much more challenging. Byzantine failures, with nodes behaving arbitrarily, are excluded from the present discussion.

We assume that nodes are connected through an existing routed network, such as the Internet, where every node can potentially communicate with every other node. To actually communicate, a node has to know the identifiers of a set of other nodes (its *neighbors*). This neighborhood relation over the nodes defines the topology of the *overlay network*. Given the large scale and the dynamism of our envisioned system, neighborhoods are typically limited to small subsets of the entire network. The neighbors of a node (and, thus, the overlay topology) can change dynamically.

7.3 The T-CHORD protocol

Let us now describe the proposed algorithms. In this section we will heavily build on algorithms and concepts introduced in Chapter 6.

7.3.1 A Brief Introduction to Chord

CHORD is an example of a key-based overlay routing protocol. In such protocols, subsets of the key space are assigned to nodes, and each node has a routing table that it uses to route messages addressed by a specific key towards the node that is responsible for that key. These routing protocols are used as a component in the implementation of the *distributed hash table* abstraction, where (key, object) pairs are stored over a decentralized collection of nodes and retrieved through the routing protocol.

We provide a simplified description of CHORD, necessary to understand T-CHORD. Nodes are assigned random t-bit IDs; keys are taken from the same space. The ID length t must be large enough to make the probability of two nodes or two keys having the same ID negligible. Nodes are ordered in an sorted ring as described in Section 6.7.3. The way this ring is constructed naturally inspires a *follows* relation over the entire ID (and key) space: we say that a follows b if $(a - b + 2^t) \mod 2^t < 2^{t-1}$; otherwise, a *precedes* b. We also define a notion of distance, again, inspired by the sorted ring, as follows: $d(a, b) = \min(|a - b|, 2^t - |a - b|)$. The *successor* of an arbitrary number i (that is, not necessarily existing node ID) is the node with the smallest ID that follows i, as defined above. We denote the successor of i by $succ_1(i)$. The concepts of predecessor, j^{th} successor, and j^{th} predecessor are defined similarly. Key k is under the responsibility of node $succ_1(k)$.

Each node maintains a routing table that has two parts: *leaves* and *fingers*. Leaves define an r-regular ring lattice, where each node n is connected to its r nearest successors $succ_1(n) \dots succ_r(n)$ in the sorted ring. Fingers are long range links: for each node n, its j^{th} finger is defined as $succ_1(n + 2^j)$, with $j \in [0, t - 1]$. Routing in CHORD works by forwarding messages following the successor direction: when receiving a message targeted at key k, a node n forwards it to its leaf or finger that precedes (or is equal to) and is closest to $succ_1(k)$, the intended recipient of the message.

Due to the fingers, the number of nodes that need to be traversed to reach a destination node is $O(\log N)$ (with high probability), where N is the size of the network [83]. Leaves, on the other hand, are used to improve the probability of delivering a message in case of failures, and to avoid that the ring can be broken into disjoint partitions.

7.3.2 T-CHORD

In the context of CHORD, our overlay construction problem translates to initializing the routing tables of all nodes simultaneously from scratch. The existing join protocol is not designed to handle the massive concurrency involved in a jump-starting process, when all the nodes are trying to join at the same time [83]. On the other hand, naive approaches where nodes are forced to join the overlay in some specified order results in at least linear time needed to construct the network (not to mention the serious problem of synchronizing the operations).

For constructing the leaf set and the fingers simultaneously, we apply T-MAN with an appropriate ranking method. As usual, we use node ID-s as node profiles. The ranking method we adopt is simply the ranking method of SORTED RING as seen in Chapter 6. At any time, the actual leaf and finger sets are then constructed by each node locally from nodes in their current view. Note, that the view is not bounded, so the node descriptors that were received in the initial (non-converged) cycles are available as well. These nodes are not useful for the leaf set (that defines the ring), however, they are crucial for the fingers, that represent shortcuts in the ring.

As of starting and termination, we experiment with both synchronous and realistic starting and termination policies. The realistic policies are those that were described in Chapter 6.

7.3.3 T-CHORD-PROX: Network Proximity

At a node n, for an exponent $j \in [1, t-1]$, several nodes in the current view may belong to the finger range $F_j = [n + 2^{j-1} \mod 2^t, n + 2^j - 1 \mod 2^t]$. In T-CHORD, the finger nearest to n with respect to the ID space was selected among them, according to the convention applied in the original CHORD. However, exploiting this degree of freedom would allow us to optimize for message latency (a key measure of performance in routing) and select the fastest possible finger that falls in the interval. This would enable the construction of low-latency routing paths between nodes, improving the overall routing performance of the network.

Inspired by this insight, we propose T-CHORD-PROX, a variant of T-CHORD based on proximity. The protocol is the same as before, however, when constructing the finger table, for all finger exponents j T-CHORD-PROX picks p nodes at random from F_j (or the entire F_j set, if its size is less then or equal to the parameter p), and measures the latency by sending *distance probes* to them. A distance probe can be implemented as a simple ping-pong exchange, for example. This simple protocol builds a routing network that results in a number of hops similar to the original CHORD, but outperforms it in terms of latency.

7.4 Experimental Results

We performed extensive simulation experiments in order to compare the jump-started overlay to the perfect CHORD topology, and to characterize the scalability and robustness of our protocols. All of the experimental results were obtained using PEERSIM [66].

7.4.1 Experimental Settings

By default, in all experiments all nodes are initialized with a random view obtained from the NEWSCAST protocol (see Section 2.2.4). Subsequently, T-MAN is run to create an ordered ring, and to collect long range links as well. When T-MAN reaches a pre-specified number of cycles, each node runs T-CHORD locally to extract its routing tables from the T-MAN view, creating the CHORD topology. We note that in Section 7.4.6 we deviate from this default in order to analyze practical starting and termination mechanisms.

We focus on the *routing performance* of the obtained overlay. Three routing metrics have been taken into consideration. *Hop count* is the number of nodes that are traversed by a message to reach its destination. In case of failures, message timeouts (*failed hops*) are counted separately. *Delivery delay* measures the time needed to reach the destination. Our latency model is based on the King dataset [137], that provides end-to-end latency measurements for a set of 1740 routers. Each node is attached through a 1ms link to a randomly selected router [81]. In case of failures, a time equal to twice the latency is added to the total delay in order to simulate timeouts. *Loss rate* is the fraction of messages that do not reach the destination node.

Since our goal is to jump-start CHORD, the baseline routing performance is defined by the perfect CHORD topology over the same set of nodes. We construct this topology

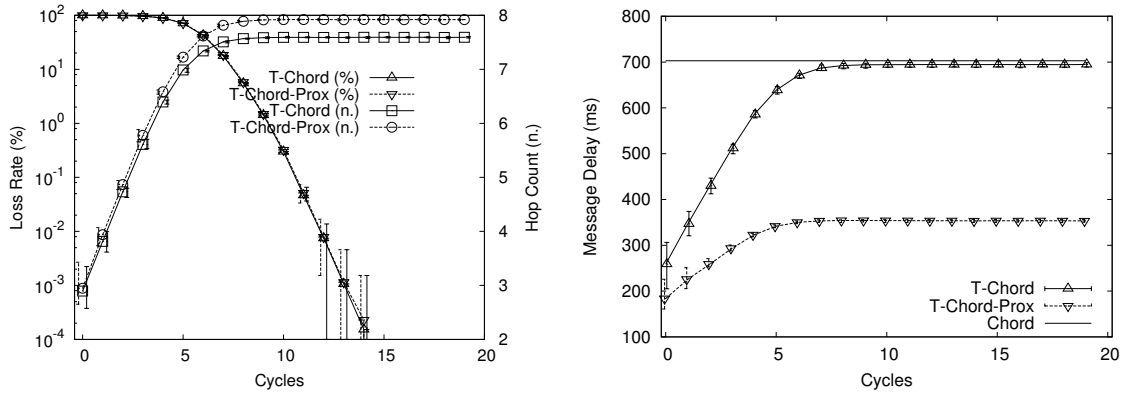


Figure 7.1: Loss rate, hop count and message delay as a function of the number of T-MAN cycles executed

off-line, using the specification of the CHORD protocol, and we compare the performance of this ideal topology with the ones generated by T-CHORD. We emphasize again that our goal is not to develop a novel routing mechanism or a new structured overlay: our goal is to create a CHORD topology efficiently from scratch.

Besides routing performance, we also need to measure communication overhead for building the topology. In case of T-CHORD without proximity, communication costs are given just by T-MAN exchanges. Given the periodic nature of T-MAN, these costs can be easily computed: each T-MAN node sends one message and receives one message on the average per cycle, with m descriptors included in each message. T-MAN is run for $O(\log N)$ cycles. In T-CHORD-PROX, the cost of latency probes must also be considered.

Unless stated otherwise, all figures are based on the following parameters: network size $N = 2^{16}$ nodes, message size $m = 10$, size of the leaf set in the CHORD target topology $r = 5$, maximum number of probes per routing table entry $p = 5$. In all figures, 20 individual experiments were performed. Average values for each of the metrics are shown; error bars are used to show minimum and maximum values among the experiments (standard deviation is often too small to be visualized). To aid the visualization, some of the bars are slightly shifted horizontally.

7.4.2 Convergence

The routing performance of the topologies obtained by T-CHORD depends on the number of T-MAN cycles executed before the routing tables are built. In particular, the ring must be completed in order to guarantee the correct delivery of all messages. This is illustrated in Figure 7.1, where the loss rate and the observed hop count for T-CHORD and T-CHORD-PROX are shown as a function of the number of T-MAN cycles that have been run. Initially, all messages are lost: local views contain only random nodes, so the routing algorithm is unable to deliver messages. The loss rate rapidly decreases, however, reaching 0 after only 14 cycles. At that point, the leaf ring is completely formed in all our experiments. Note that the curves for T-CHORD and T-CHORD-PROX overlap almost completely.

Regarding hop counts, the results confirm that the quality of the routing tables stabilizes after few cycles, for both versions of T-CHORD. Message delay follows a similar behavior, except that T-CHORD-PROX shows a significant improvement. The increasing tendency of the hop count curves is explained by the fact that in the beginning, in spite of the low quality overlay, a few messages reach their destination "by chance" in a few hops,

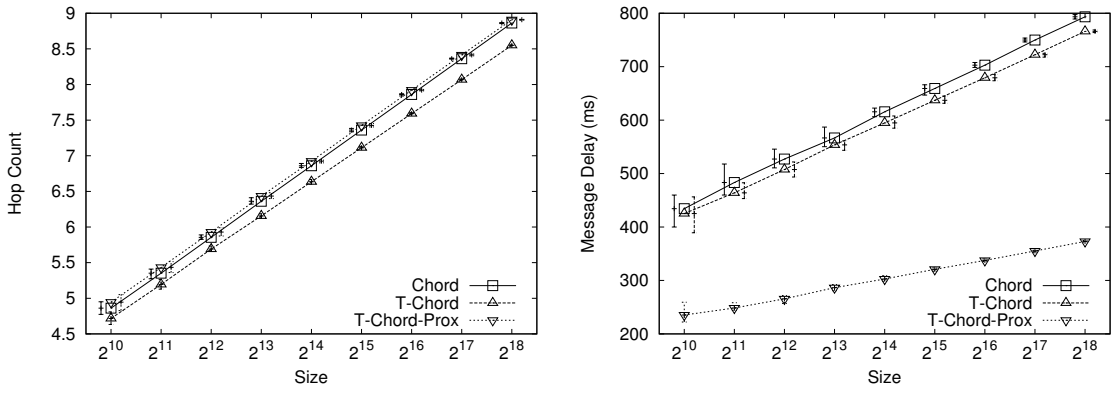


Figure 7.2: Average hop count and message delay as a function of network size

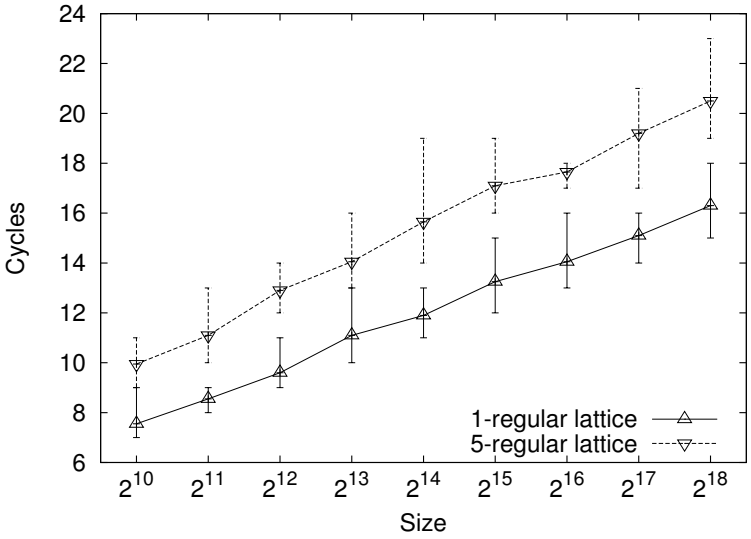


Figure 7.3: Convergence time (cycles) as a function of network size

while most of the messages are lost.

7.4.3 Scalability

The experiments discussed so far were run in a network with a fixed size (2^{16} nodes). To assess the scalability of T-CHORD, Figure 7.2 plots measurements against network size varying in the range $[2^{10}, 2^{18}]$. Results for the ideal CHORD topology are also shown. All algorithms scale logarithmically with size.

Quite interestingly, T-CHORD performs slightly better than CHORD. Regarding hop count, this is explained by the fact that the distance of the longest fingers tend to be larger in our case (due to not strictly satisfying the CHORD specification), which speeds up reaching the destination node if it resides in the most distant half of the ring. Regarding message delay, as expected, T-CHORD-PROX outperforms both T-CHORD and CHORD, due to its latency-optimized set of fingers. To obtain such performance, T-CHORD-PROX pays a price in terms of latency probes. In this experimental setting, with parameter p set to 5, we have observed a total number of probes per node scaling logarithmically from 45 (for $N = 2^{10}$) to 77 (for $N = 2^{18}$). This is expected, as the number of expected different finger entries per node is $O(\log N)$ [83]. These values can be tuned by varying the p parameter.

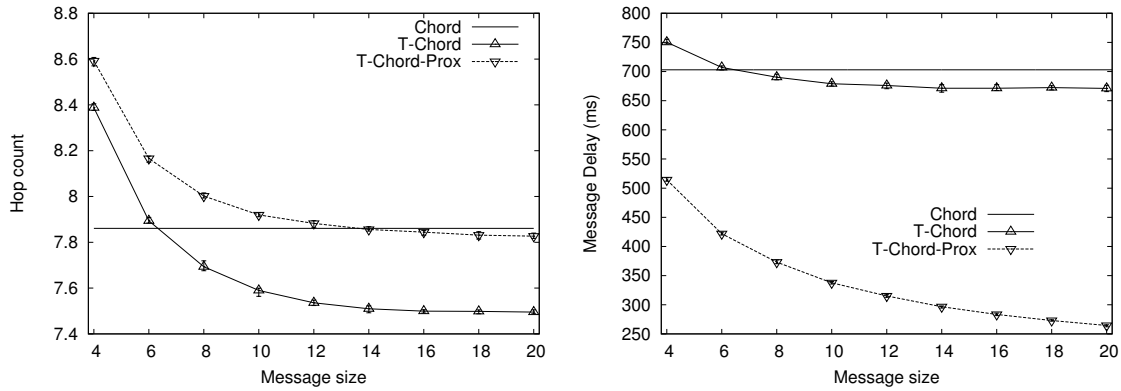


Figure 7.4: Hop count and message delay as a function of message size m

Figure 7.3 plots the number of cycles needed to obtain the 1-regular lattice (the ring), sufficient to guarantee the consistent routing of messages (absence of message losses) [83], and the r-regular lattice used to provide additional fault-tolerance. In both cases, the convergence is obtained in a logarithmic number of cycles. Note that the actual execution time of the protocol depends on the length of a cycle, which is a parameter of the protocol. Based on results in Chapter 6, a cycle length of 1-2 seconds is very reasonable. Considering the results, we can conclude that any practical network can very safely be constructed in less then a minute (30 cycles).

Finally, as part of our measurements regarding scalability, let us consider the storage complexity of T-MAN. In Section 6.6.4 we argued that storage complexity is $O(\log N)$ per node. This is why we do not have an upper bound on the view size. Indeed, in our simulation experiments, the average amount of descriptors discovered during the execution ranges from as little as 70 ($N = 2^{10}$) to 140 ($N = 2^{18}$).

7.4.4 Parameters

To evaluate the impact of the T-MAN message size (m) on the routing performance of our algorithm, we performed the simulations shown in Figure 7.4. For message size m we set the size of the CHORD leaf set to be $r = m/2$. The plots show that good results are obtained even when using small message size, although it must be noted that in the case of $m = 4$, approximately 0.6% of the messages are not delivered to their destination.

7.4.5 Robustness

To test robustness, we have considered two different failure models: *crash* and *churn*. In the former, failures are catastrophic: a given percentage of nodes are suddenly removed from the completed CHORD network. In the latter, the same percentage of nodes are removed during the execution of T-CHORD, evenly distributed over time.

The two models play different roles in our analysis. The crash model is the only one applicable to the ideal CHORD network that we use for comparison, since we build it off-line, without using the actual CHORD maintenance protocol. We use this model to obtain a lower bound for routing performance. In the churn model, on the other hand, failures influence the execution of T-MAN; we use this model to show that our algorithm can indeed survive failures during its execution.

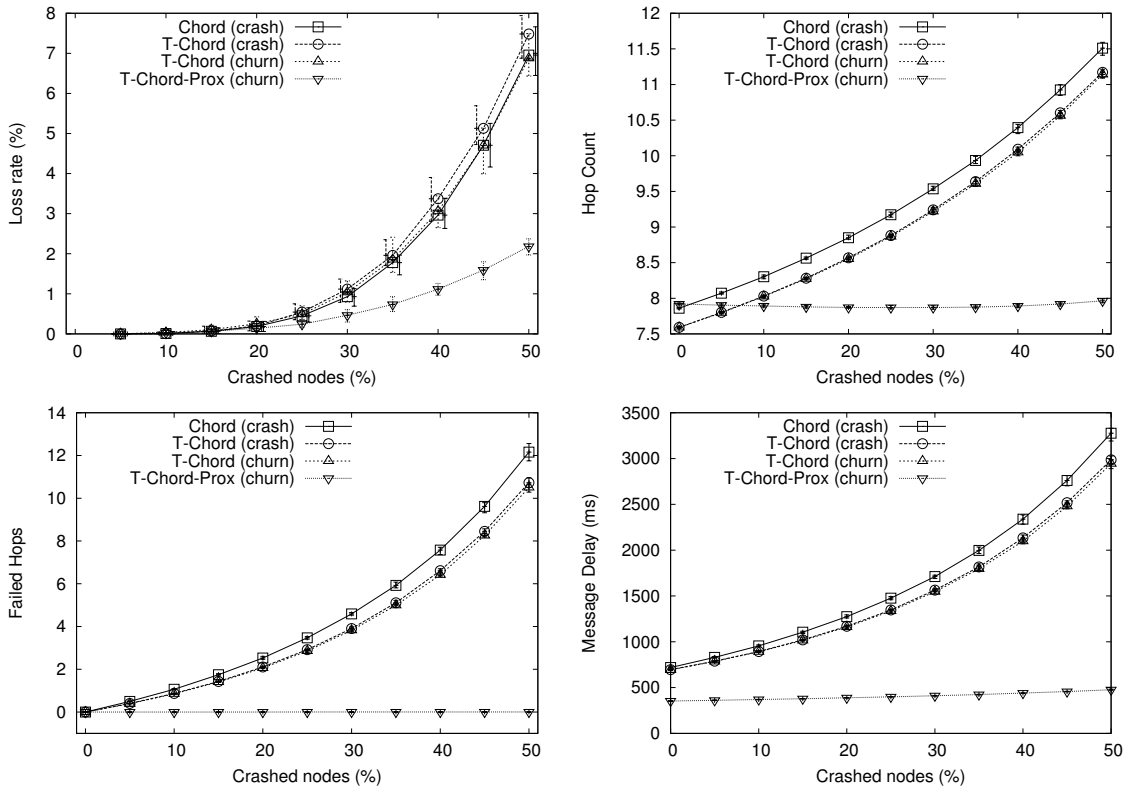


Figure 7.5: Loss rate, hop count, failed hops, and message delay under different failure scenarios

It is important to note that a direct comparison between the results of T-CHORD-PROX and the other results is not fair. T-CHORD-PROX probes nodes for latency before inserting them in the finger set, which means that only a few fingers (the ones that fail in the period after the probing) are down when the routing performance is evaluated.

We have simulated an increasing percentage of nodes removed in a network of size 2^{16}, with T-MAN running for 20 cycles. The results are presented in Figure 7.5. Once again, our routing metrics show that the topology obtained by T-CHORD without proximity is comparable to the ideal CHORD topology, in both the crash and the churn models.

It is interesting to compare the simulated churn rate with the churn rate observed in deployed P2P networks [141]. In the worst case, the churn rate corresponds to 50% divided by 20 cycles, i.e. 2.5% per cycle. A cycle length of 2 seconds (a perfectly reasonable choice that enables the construction of a 2^{16} topology in less than a minute) corresponds to 0.0125 failures per node per second, two orders of magnitude larger than the rates observed in deployed networks (around 10^{-4} failures per node per second ([141])).

7.4.6 Starting and Termination

So far we have experimented with the protocol assuming that the startup is synchronized, and that termination is based on a pre-determined number of cycles. Here we add to the protocol the startup and termination mechanisms described in Chapter 6. We will now characterize time in terms of seconds and not cycles, due to the more fine-grained nature of the simulations. We set $\Delta = 1$. In the experiments here $m = 10$, however, we build leaf sets of size $r = 10$ and not $r = 5$ as before, so convergence is somewhat slower even

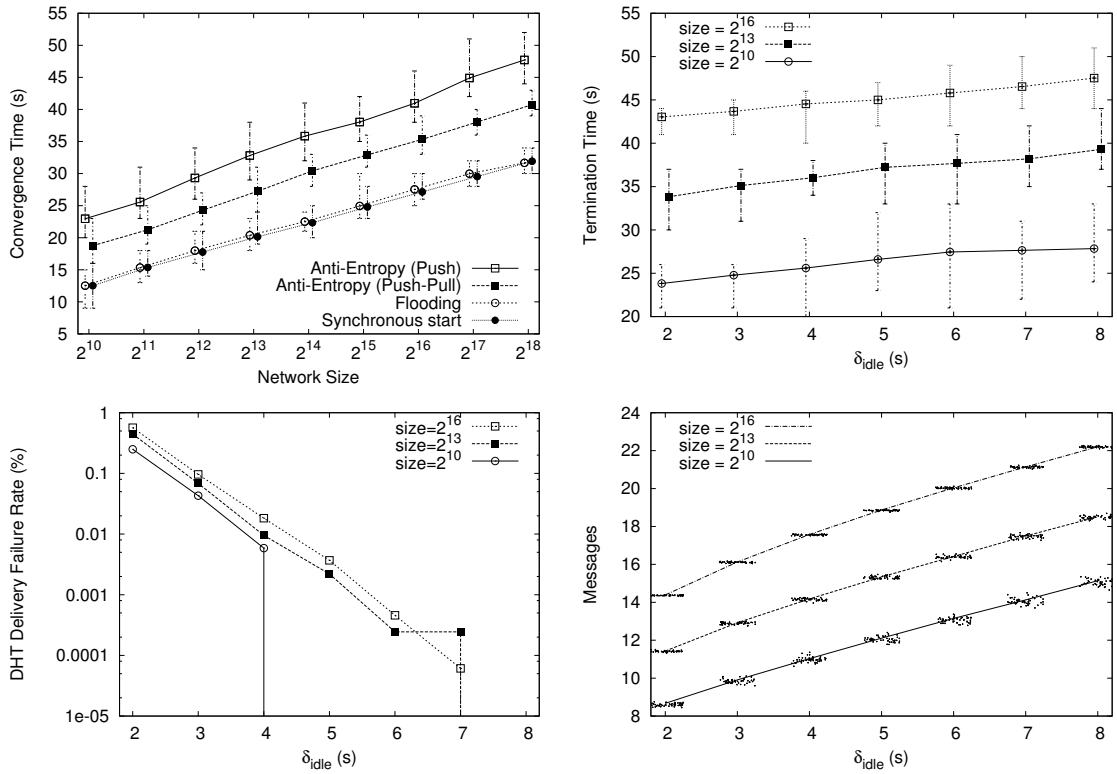


Figure 7.6: The effect of parameters affecting starting and termination.

for synchronous startup.

Figure 7.6 contains the results of this set of experiments. The upper left plot shows the convergence time for different starting protocols and for a variable network size. The relative convergence time of the different mechanisms is similar to what can be seen in Figure 6.9.

The upper right plot presents termination times for different values of parameter δ_{idle}. For small values of δ_{idle} and for large networks, we found that the protocol never reaches convergence. Nevertheless, the lower left plot shows that even for small values of δ_{idle}, the number of messages never delivered to the correct destination is smaller than 1%, which means that the obtained overlay is a good approximation of CHORD. However, for $\delta_{idle} = 8$, all our test runs resulted in 100% successful message delivery, so we adopt this value for the protocol. The slight disadvantage is a larger number of messages exchanged and a slower termination time.

Finally, the lower right plot shows the average number of messages sent by a node in the network until termination. This is significantly lower than the termination time, which could be expected based on our findings discussed before (see Figure 6.12), namely that most of the nodes terminate much earlier than the global termination time.

7.5 Related Work

Bootstrapping structured overlays is somewhat under-emphasized in comparison with other research topics. Existing proposals have assumed networks that are already formed, or networks that grow progressively, using the native join protocol. The discovery of the node to join may be facilitated either by a central (well-known) node, or through a

universal ring, a shared overlay providing discovery and deployment services [142].

Join protocols enable a new node to find its position inside the structured topology [81, 83]. For example, the single-join protocol of CHORD requires a node to locate its position inside the ring, and then to locate each of its $O(\log N)$ distinct fingers [83]. Since both operations require $O(\log N)$ hops (messages), the cost of a single-join is $O(\log^2 N)$.

This aggressive protocol is superseded by a light-weight one that can support concurrent joins. In this case, nodes just find their position in the ring (with a $O(\log N)$ routing operation), while fingers are updated subsequently by a *stabilization protocol*. The protocol is efficient "*... unless a tremendous number of nodes joins the system.*" [83], in which case the updating rate of fingers is not sufficient and routing requires a linear number of hops. In comparison, our approach builds the topology in $O(\log N)$ cycles, with two messages sent and two messages received per node per cycle, with each message being a collection of m 128-bit IDs.

The problem of bootstrapping an overlay topology has started recently to gain interest from the research community. Angluin et al. [129] propose an asynchronous algorithm whose goal is to build a linked list of nodes sorted by their identifiers. Their approach is based on binary search trees that are built in $O(WlogN)$ time, where W is the length of node identifiers. On comparison, our approach builds the ring in $O(logN)$ time, independently from the size of identifiers. Furthermore, our approach can deal with high level of churn, while churn has not been considered in [129]. Aberer et al. [127] propose a mechanism for bootstrapping a P-Grid topology in $O(log^2 N)$ time.

Finally, Voulgaris and van Steen [126] propose an epidemic protocol with a similar goal: jump-starting Pastry. However, their proposal is rather expensive: it requires running $O(\log K)$ instances of a modified NEWSCAST protocol [6] in parallel (where K is the size of the ID space), and it does not take latency into account. Besides, it is highly specific to Pastry, whereas our approach, being based on T-MAN, that is able to evolve a wide range of topologies, is potentially more generic. Indeed, we already have preliminary results for building Pastry as well, through an XOR-based ranking function for T-MAN, with costs similar to T-CHORD.

7.6 Conclusions

We have addressed the problem of jump-starting a popular structured overlay, CHORD, from scratch. The proposed protocols, T-CHORD and T-CHORD-PROX are scalable, light-weight and robust, and can be applied to scenarios (such as Grids [140] and large-scale testbeds like Planet-Lab [95]), where the overlay infrastructure needs to be built from the ground up as quickly and efficiently as possible.

Although here we targeted CHORD, our approach is more general, and it can be applied to other overlay protocols as well. Chapter 8 will contain an example where we build a prefix-table based overlay.

Chapter 8

Towards a Generic Bootstrapping Service

In this last chapter, we present another application of T-MAN, namely the bootstrapping of prefix-based structured overlay networks such as PASTRY. In addition, we also propose a modular approach to combine several gossip components, and we argue that bootstrapping different overlay networks is an important service in this architecture.

The novel application scenarios for P2P systems that are supported by this *bootstrapping service* include merging and splitting of large networks, or multiplexing relatively short-lived applications over a pool of shared resources. In such scenarios, the architecture needs to be quickly and efficiently (re)generated frequently, often from scratch. We propose the bootstrapping service abstraction as a solution to this problem. We present an instance of the service that can jump-start any prefix-table based routing substrate quickly, cheaply and reliably from scratch. We experimentally analyze the proposed bootstrapping service, demonstrating its scalability and robustness.

8.1 Introduction

Structured overlay networks are increasingly seen as a key layer (or service) in peer-to-peer (P2P) systems, supporting a wide variety of applications. Index-based lookup is generally considered to be a "bottom" layer (e.g., [135, 142]), based on the assumption that the life cycle of supported systems is similar to grassroots file sharing networks: there exists at least one functional network, membership can change due to churn, and the network size can also fluctuate, but relatively smoothly. Join operations are assumed to be uncorrelated. Most simulation and analytical studies also reflect these assumptions, since they are often based on traces collected from real file sharing networks.

While this scenario may be appropriate for many important applications, we believe that overlay networks can be important design abstractions in radically different scenarios that have not yet been considered by the P2P research community. In particular, *massive joins* to a large overlay network are not supported by known protocols very well, and many protocols have trouble dealing with *massive departures* as well. Other related scenarios that are important yet under-emphasized include *bootstrapping* a large network from scratch, *merging* two or more networks, *splitting* a large network into several pieces, and *recovering from catastrophic failure*.

If these scenarios were to be supported efficiently, we could build a fully open and flexible computing infrastructure that points well beyond current applications. We envi-

sion scenarios that involve (virtual) organizations with (possibly) large pools of resources organized in overlay networks. We want to allow these overlay networks to freely and flexibly merge with and split from networks of other organizations on demand, and we want to admit allocation (or sale) of pools of resources for relatively short periods to users who could then build their own infrastructures on demand and abandon them when they are done. This vision is in line with current efforts to enhance the flexibility of Grid infrastructures using P2P technology [143].

To support the above vision, we propose a P2P architecture with two main components: the *peer sampling service* and a dedicated *bootstrapping service*. Merging several large networks or starting an application from scratch within its time-slice are unusual and radical events that many existing P2P protocols are not designed to cope with. To provide a reliable platform in the face of massive joins and departures, we propose the *peer sampling service* (see Section 2) as a lightweight bottom-most layer of our P2P architecture. The bootstrapping service is then built on top of this peer sampling service. In the proposed architecture, large collections of resources can be readily aggregated into global structured overlays rapidly and efficiently. This then allows the use of existing, well-tuned protocols without modification to maintain the overlays once they have been formed. As a concrete example of the bootstrapping service, we present a novel protocol that can efficiently build prefix-based overlay routing substrates such as Pastry [81], Kademlia [144], Tapestry [145] and Bamboo [51] from scratch.

Considering related work in the area, massive joins to already running overlays have been addressed previously (e.g., [134, 135]) proposing a form of periodic repair mechanism for maintaining the leaf set, not unlike the one presented here. More recently the bootstrapping problem has been addressed as well, focusing on specific overlays [16, 127, 128]. Our contribution with respect to related work is twofold. First, we propose an *architecture* that can support a protocol that jump-starts an entire overlay *from scratch*. Our protocol is independent of the protocol that manages the routing substrate: we singled out the abstract bootstrap service as an important architectural component. Second, our protocol is efficient and lightweight, and supports overlays based on *prefix-tables* and leaf sets.

The outline of this chapter is as follows. Section 8.2 presents the architecture to support the scenarios mentioned above. Section 8.3 describes the protocol implementing the bootstrapping of routing substrates, while Section 8.4 presents experimental results. Finally, Section 8.5 concludes the chapter.

8.2 The Architecture

Our ultimate goal is to design a P2P architecture that allows for large pools of resources to behave almost like a liquid substance: it should be possible to merge large pools, or split existing pools into several pieces easily. Furthermore, it should be possible to bootstrap potentially complex architectures on top of these liquid pools of resources quickly on demand.

The architecture is outlined in Figure 8.1. The lowest layer, the peer sampling service, implicitly defines a group abstraction by allowing higher layers to obtain addresses of random samples from the actual set of participating peers; even shortly after massive joins or catastrophic failures. The basic idea of the architecture is that we require *only* this lowest layer to be liquid, that is, persistent to the radical scenarios we described, and we propose to build all other overlays on demand. In other words, the sampling service

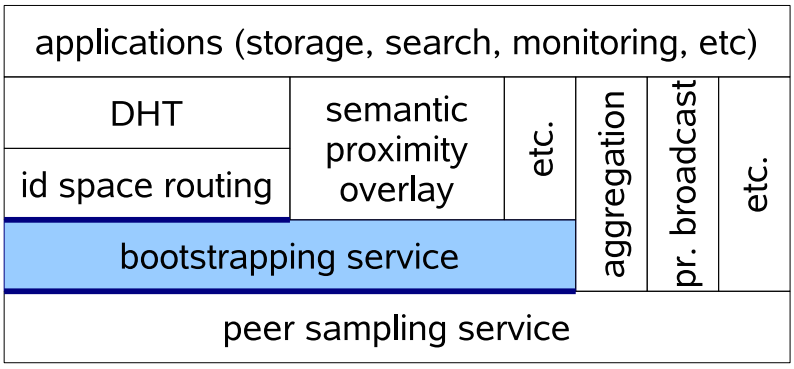


Figure 8.1: The layers of the proposed architecture. The highlighted part is discussed in this chapter.

functions as a last resort that provides a very basic, but at the same time extremely robust service, which is sufficient to enable jump starting or recovering all higher layers of the architecture.

As shown in Figure 8.1, the architecture supports other components in addition to structured overlays. For example, a number of components rely only on random samples, like probabilistic broadcast (gossip) or aggregation (see Chapter 3). The architecture can also support other overlays, such as proximity based ones (see Chapter 6).

The bottom layer of the proposed P2P architecture is the peer sampling service (see Chapter 2). Due to its low cost, extreme robustness and minimal assumptions, gossip based peer sampling protocols are an ideal bottom layer that makes the bootstrap service feasible. The sampling service is useful (and, in fact, sufficient) for gossip-based protocols that are based on sending information periodically to random peers. In this chapter, as usual, we again use the NEWSCAST protocol in our experiments (see Section 2.2.4).

8.3 Bootstrapping Prefix Tables

As argued earlier, our architecture crucially relies on the existence of a lightweight and efficient implementation of the bootstrapping service, that in turn relies on peer sampling. Here we develop a protocol that fulfills these requirements. We have already addressed bootstrapping CHORD in Chapter 7 based on a sorted ring, and additional fingers that are defined based on distance in the ID space. However, an important alternative design decision of DHT-s is applying *prefix-based routing tables*, which have some important advantages, such as independence of ID distribution, but which are a significantly different task to build and maintain. The protocol that we present here constructs prefix-based routing tables at all participating nodes simultaneously, and from scratch. The key idea is similar to that of T-CHORD; nodes build a sorted ring, and during the process they collect entries to fill the prefix tables at all nodes.

The prefix table is defined as follows. We assume that all nodes have unique numeric IDs. An ID is represented as a sequence of digits in base 2^b—each digit is encoded as a b-bit number. The prefix table of a given node contains IDs that belong to different types based on the length of the common prefix with the node's own ID. The types are defined by a pair (i, j), where i is the length (in base 2^b digits) of the longest common prefix of the ID and the node's own ID, and j is the actual value of first differing (base 2^b) digit. For each entry type (i, j) the table contains up to k alternative IDs. (Note that

it is possible that there are less than k node IDs with the desired prefix and digit among the participating nodes, in which case we cannot fill all the k slots; hence k is only an upper bound.) Many overlay routing substrates are based on this prefix table: for example Pastry [81], Kademlia [144], Tapestry [145] and Bamboo [51]. Using the constructed prefix tables and the leaf sets (that define the sorted ring), the routing tables of all these networks can be bootstrapped.

To sum up, each node has a prefix table and a leaf set to fill, and the leaf set is being evolved to contain the nearest neighbors in the sorted ring of node IDs. Unlike in the case of T-CHORD, here the leaf set is symmetric, so constructing the leaf can be achieved using SORTED RING directly. The size of the leaf set is denoted by m. The protocol we propose is similar to T-CHORD in that it is an instance of SORTED RING with some modifications. Like in T-CHORD, the prefix table entries are constantly being filled using any new information that is arriving in the incoming messages.

The main new idea w.r.t. T-CHORD is that the gradually improving prefix table is fed back into the ring building process, so that the two components mutually boost each other. That is—although the ring-building process fills in most of the entries in the prefix tables as a side-effect—the prefix tables can already fulfill a kind of routing function before being completed, just like in DHT-s. Especially in the end phase, when most of the nodes have found their place in the ring, but a few still have an incorrect neighborhood, the gossip mechanism of T-MAN and the almost complete prefix tables together can help these last nodes find their correct neighborhood quickly, essentially as if they were routed by the routing substrate under construction.

To implement this idea, we modify method TOSEND() in Algorithm 13, which is responsible for generating a set of node descriptors to be sent to the peer node. Knowing the ID of the peer, the method optimizes the information to be sent as follows. First it takes the union of the leaf set, r random samples taken from the sampling service, the current prefix table, and its own descriptor (in other words, all locally available information). It applies the ranking function to this set, and keeps the first m entries. In addition, it adds to the message all node descriptors that are potentially useful for the peer for its prefix table (i.e., have a common prefix with the peer ID). The size of this additional part is not fixed but is bounded by the size of the full prefix table, and usually is smaller in practice.

Let us summarize the parameters specific to the protocol. The prefix table is defined by b (the number of bits in a digit) and k, the number of alternatives entries to be stored for all the specific prefix types. The size of the leaf set is m. Finally, r is the number of random samples used for improving the messages to be sent. Note that these samples are "free" (if r is not too large) since the generic peer sampling layer is assumed to function independently of the bootstrapping service.

8.4 Simulation Results

Both the sampling service and the bootstrapping service were implemented for the PEERSIM simulator [66]. We focus on two aspects of the protocol: scalability and fault tolerance. To this end, we fix all the parameters of the protocol, except the network size and failure model. In our simulations IDs are 64-bit integers. Although typical definitions of the ID space consider 128-bit integers, using only 64 bits for our simulations is not limiting since the length of the largest common prefix is much less than 64 bits for all node pairs in networks of any practical size. The extra bits play no role in this protocol.

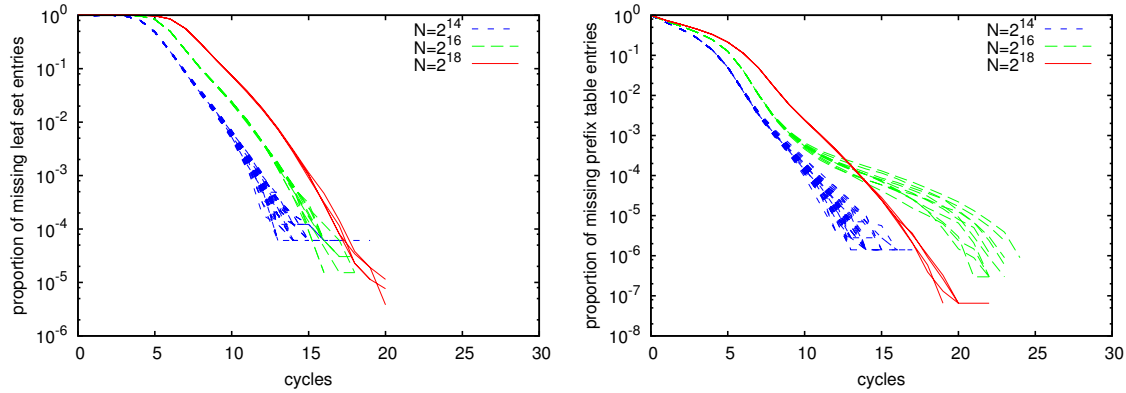


Figure 8.2: Results in the absence of failures. When a curve ends, the corresponding tables are perfect at all nodes.

The parameters of the prefix table were chosen to match common settings: $b = 4$ and $k = 3$. For networks that do not require multiple alternatives of a given table entry, setting $k > 1$ is still useful because it allows for optimizing the routes according to proximity as we did in the case of T-CHORD-PROX. The leaf set size was $m = 20$ and the parameter r was set to be 30. We experimented with network sizes (N) of 2^{14}, 2^{16} and 2^{18} nodes.

Method SELECTPEER() of T-MAN uses parameter $\psi = m/2$ (a peer is selected from the $m/2$ highest ranking nodes) and no tabu list is used. The startup technique we use is flooding, as described in Chapter 6. The protocol is then run until the *perfect* leaf sets and prefix tables are found at all nodes, based on the actual set of IDs in the network. This cannot be decided locally, and indeed, the protocol has no stopping criterion. However, since our protocol is cheap and needs only a small number of iterations, in practice, after initialization it can be run simply for a fixed number of cycles that are known to be sufficient for convergence.

To test scalability, in the first set of experiments (shown in Figure 8.2) there are no failures and all messages are delivered reliably. For network sizes 2^{14}, 2^{16} and 2^{18}, we performed 50, 10 and 4 independent experiments, respectively. The plots show the results of each individual experiment, ending when perfect convergence is obtained.

From the left plot of Figure 8.2 we observe that the time required to reach a desired quality of the leaf sets increases by an additive constant despite a four-fold increase in the network size. This is a strong indication that the time needed for convergence is logarithmic in network size. In addition to being logarithmic, the actual convergence times are also rather small. Convergence of the leaf sets clearly follows an exponential behavior.

The convergence of the prefix tables is rather surprising (right plot of Figure 8.2): the network of 2^{18} nodes converges *faster* in the final phase than a network that is four times smaller, with the same parameters. Note that in this final phase, the vast majority of the entries are already available (less than 1 out of 1000 entries are missing). This slight difference has to do with the scarcity of suitable IDs for the remaining positions to fill.

In the second set of experiments we tested the robustness of our protocol by dropping messages with a uniform probability (Figure 8.3). This failure model is appropriate for study because we designed the protocol with a cheap, unreliable transport layer in mind (UDP). The drop probability was chosen to be 20%, which is unrealistically large. Since the protocol is based on message-answer pairs, if the first message is dropped, then the answer is not sent either. Taking this effect into account, elementary calculation shows

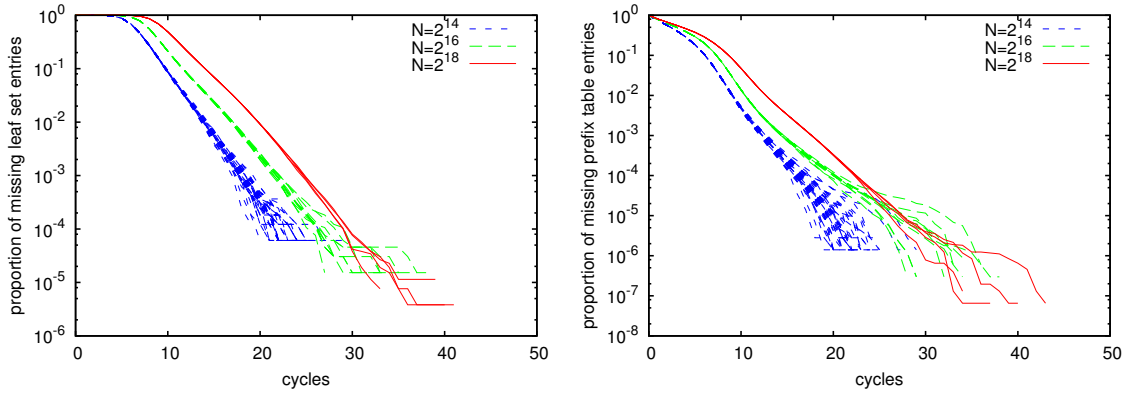


Figure 8.3: Results with 20% of the messages dropped. When a curve ends, the corresponding tables are perfect at all nodes.

that the expected overall loss of messages is 28%.

The main conclusion of these experiments is that the behavior of the protocol is very similar to the case when there are no failures, only convergence is slowed down proportionally.

The protocol is not sensitive to churn either (not shown). In short, the quality of the routing tables generated by our protocol is similar to that obtained by known routing substrates in the presence of similar churn. Furthermore, since our protocol is based on cheap UDP messages and can be completed in a small number of cycles, the effect of churn during this short time is naturally limited.

8.5 Conclusions

We proposed a P2P architecture that relies on a robust peer sampling service and a bootstrapping service. Although the functionality of the sampling service is basic, its implementation is more robust and flexible than those of currently available structured overlays. The architecture we presented here, and in particular, the bootstrapping service, bridges the robustness and flexibility of the sampling service and the functionality of structured overlays.

Based on our simulation results, the proposed instantiation of the bootstrapping service can build a *perfect* prefix table and leaf set at all nodes, in a logarithmic number of cycles, even in the presence of message delivery failures. This performance, in combination with the support of the sampling layer, enables the on-demand deployment of complex (multi-layered) P2P applications in short time-slices over large pools of shared resources, in addition to allowing large pools of resources to be merged or split temporarily. Note that (as presented in Chapter 7) the bootstrapping service can be instantiated by T-CHORD as well, in which case a finger table is produced instead of a prefix table. Also note that the finger table can also be fed back to the construction process like the prefix table in this chapter; a possibility we have not used in our presentation in Chapter 7.

Bibliography

[1] Jelasity, M.: Gossip. In Di Marzo Serugendo, G., Gleizes, M.P., Karageorgos, A., eds.: Self-Organising Software: From Natural to Artificial Adaptation. Natural Computing Series. Springer (2011) 139–162

[2] Costa, P., Gramoli, V., Jelasity, M., Jesi, G.P., Le Merrer, E., Montresor, A., Querzoni, L.: Exploring the interdisciplinary connections of gossip-based systems. ACM SIGOPS Operating Systems Review **41**(5) (2007) 51–60

[3] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., van Steen, M.: Gossip-based peer sampling. ACM Transactions on Computer Systems **25**(3) (August 2007) 8

[4] Jelasity, M., Kowalczyk, W., van Steen, M.: Newscast computing. Technical Report IR-CS-006.03, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands (November 2003)

[5] Jelasity, M., Kowalczyk, W., van Steen, M.: Newscast computing. In Enachescu, C., Filip, F.G., Iantovics, B., eds.: Advanced Computational Technologies. Romanian Academy Publishing House, Bucharest, Romania (2012) 22–44 Reprint of VU University Tech. Rep. IR-CS-006.03.

[6] Jelasity, M., Guerraoui, R., Kermarrec, A.M., van Steen, M.: The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Jacobsen, H.A., ed.: Middleware 2004. Volume 3231 of Lecture Notes in Computer Science., Springer-Verlag (2004) 79–98

[7] Tölgyesi, N., Jelasity, M.: Adaptive peer sampling with newscast. In Sips, H., Epema, D., Lin, H.X., eds.: Euro-Par 2009. Volume 5704 of Lecture Notes in Computer Science., Springer-Verlag (2009) 523–534

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

[9] Jelasity, M., Montresor, A.: Epidemic-style proactive aggregation in large overlay networks. In: Proceedings of The 24th International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, IEEE Computer Society (2004) 102–109

[10] Montresor, A., Jelasity, M., Babaoglu, O.: Robust aggregation protocols for large-scale overlay networks. In: Proceedings of The 2004 International Conference on Dependable Systems and Networks (DSN), Florence, Italy, IEEE Computer Society (2004) 19–28

[11] Jelasity, M., Canright, G., Engø-Monsen, K.: Asynchronous distributed power iteration with gossip-based normalization. In Kermarrec, A.M., Bougé, L., Priol, T., eds.: Euro-Par 2007. Volume 4641 of Lecture Notes in Computer Science., Springer-Verlag (2007) 514–525

[12] Jelasity, M., Kermarrec, A.M.: Ordered slicing of very large-scale overlay networks. In: Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing (P2P 2006), Cambridge, UK, IEEE Computer Society (September 2006) 117–124

[13] Montresor, A., Jelasity, M., Babaoglu, O.: Decentralized ranking in large-scale overlay networks. In: Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008), IEEE Computer Society (2008) 208–213

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

[15] Jelasity, M., Babaoglu, O.: T-Man: Gossip-based overlay topology management. In Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F., eds.: Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers. Volume 3910 of Lecture Notes in Computer Science., Springer-Verlag (2006) 1–15

[16] Montresor, A., Jelasity, M., Babaoglu, O.: Chord on demand. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P 2005), Konstanz, Germany, IEEE Computer Society (August 2005) 87–94

[17] Jelasity, M., Montresor, A., Babaoglu, O.: The bootstrapping service. In: Proceedings of the 26th International Conference on Distributed Computing Systems Workshops (ICDCS WORKSHOPS), Lisboa, Portugal, IEEE Computer Society (2006) International Workshop on Dynamic Distributed Systems (IWDDS).

[18] Dunbar, R.: Grooming, Gossip, and the Evolution of Language. Harvard University Press (1998)

[19] Kimmel, A.J.: Rumors and Rumor Control: A Manager's Guide to Understanding and Combatting Rumors. Lawrence Erlbaum Associates (2003)

[20] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), Vancouver, British Columbia, Canada, ACM Press (August 1987) 1–12

[21] Pittel, B.: On spreading a rumor. SIAM Journal on Applied Mathematics **47**(1) (February 1987) 213–223

[22] Karp, R., Schindelhauer, C., Shenker, S., Vöcking, B.: Randomized rumor spreading. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00), Washington, DC, USA, IEEE Computer Society (2000) 565–574

[23] Bailey, N.T.J.: The mathematical theory of infectious diseases and its applications. second edn. Griffin, London (1975)

[24] Kempe, D., Kleinberg, J., Demers, A.: Spatial gossip and resource location protocols. Journal of the ACM **51**(6) (2004) 943–967

[25] Hand, E.: Head in the clouds. Nature **449** (October 2007) 963

[26] Lohr, S.: Google and i.b.m. join in 'cloud computing' research. The New York Times (Oct 8 2007)

[27] Amazon Web Services: http://aws.amazon.com

[28] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP'07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, New York, NY, USA, ACM (2007) 205–220

[29] Kempe, D., McSherry, F.: A decentralized algorithm for spectral analysis. In: Proceedings of the 36th ACM Symposium on Theory of Computing (STOC'04), New York, NY, USA, ACM (2004) 561–568

[30] Xiao, L., Boyd, S., Lall, S.: A scheme for robust distributed sensor fusion based on average consensus. In: IPSN'05: Proceedings of the 4th international symposium on Information processing in sensor networks, Piscataway, NJ, USA, IEEE Press (2005) 9

[31] Babaoglu, O., Canright, G., Deutsch, A., Di Caro, G.A., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T.: Design patterns from biology for distributed computing. ACM Transactions on Autonomous and Adaptive Systems **1**(1) (September 2006) 26–66

[32] Kermarrec, A.M., van Steen, M., eds.: ACM SIGOPS Operating Systems Review 41. ACM (October 2007) Special issue on Gossip-Based Networking.

[33] Johansen, H., Allavena, A., van Renesse, R.: Fireflies: scalable support for intrusion-tolerant network overlays. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06), New York, NY, USA, ACM (2006) 3–13

[34] Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03), IEEE Computer Society (2003) 482–491

[35] Mehyar, M., Spanos, D., Pongsajapan, J., Low, S.H., Murray, R.M.: Asynchronous distributed averaging on communication networks. IEEE/ACM Trans. Netw. **15**(3) (2007) 512–520

[36] Wuhib, F., Dam, M., Stadler, R., Clemm, A.: Robust monitoring of network-wide aggregates through gossiping. In: Proc. 10th IFIP/IEEE International Symposium on Integrated Management (IM 2007), Munich, Germany (May 2007) 21–25

[37] Jesus, P., Baquero, C., Almeida, P.S.: Fault-tolerant aggregation for dynamic networks. In: Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS). (November 2010) 37–43

[38] Eyal, I., Keidar, I., Rom, R.: Limosense – live monitoring in dynamic sensor networks. In Erlebach, T., Nikoletseas, S., Orponen, P., eds.: Algorithms for Sensor Systems. Volume 7111 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2012) 72–85

[39] He, W., Liu, X., Nguyen, H.V., Nahrstedt, K., Abdelzaher, T.: PDA: Privacy-preserving data aggregation for information collection. ACM Trans. Sen. Netw. **8**(1) (August 2011) 6:1–6:22

[40] Olshevsky, A., Tsitsiklis, J.N.: Convergence speed in distributed consensus and averaging. SIAM Journal on Control and Optimization **48**(1) (February 2009) 33–55

[41] Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE Transactions on Information Theory **52**(6) (2006) 2508–2530

[42] Xiao, L., Boyd, S., Kim, S.J.: Distributed average consensus with least-mean-square deviation. Journal of Parallel and Distributed Computing **67**(1) (January 2007) 33–46

[43] Lovász, L.: Random walks on graphs: A survey. In Miklós, D., Sós, V.T., Szőnyi, T., eds.: Combinatorics, Paul Erdős is Eighty. Volume 2. János Bolyai Mathematical Society, Budapest (1996) 353–398

[44] Prieto, A.G., Stadler, R.: A-gap: An adaptive protocol for continuous network monitoring with accuracy objectives. IEEE Trans. on Netw. and Serv. Manag. **4**(1) (June 2007) 2–12

[45] Birman, K.P., van Renesse, R., Vogels, W.: Scalable data fusion using astrolabe. In: Proceedings of the Fifth International Conference on Information Fusion (FUSION 2002). Volume 2. (2002) 1434–1441

[46] Bawa, M., Garcia-Molina, H., Gionis, A., Motwani, R.: Estimating aggregates on a peer-to-peer network. Technical Report 2003-24, Stanford InfoLab (April 2003)

[47] Chen, J.Y., Pandurangan, G., Xu, D.: Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis. IEEE Transactions on Parallel and Distributed Systems **17**(9) (2006) 987–1000

[48] Massoulié, L., Merrer, E.L., Kermarrec, A.M., Ganesh, A.: Peer counting and sampling in overlay networks: random walk methods. In: PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM Press (2006) 123–132

[49] Mosk-Aoyama, D., Shah, D.: Fast distributed algorithms for computing separable functions. IEEE Transactions on Information Theory **54**(7) (2008) 2997–3007

[50] Moallemi, C.C., Van Roy, B.: Consensus propagation. IEEE Transactions on Information Theory **52**(11) (2006) 4753–4766

[51] Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proceedings of the USENIX Annual Technical Conference. (June 2004)

[52] van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. In: Middleware '98, The Lake District, England, IFIP (1998) 55–70

[53] Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Transactions on Computer Systems **17**(2) (May 1999) 41–88

[54] Kowalczyk, W., Vlassis, N.: Newscast EM. In Saul, L.K., Weiss, Y., Bottou, L., eds.: 17th Advances in Neural Information Processing Systems (NIPS), Cambridge, MA, MIT Press (2005) 713–720

[55] Gupta, I., Birman, K.P., van Renesse, R.: Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. Quality and Reliability Engineering International **18**(3) (2002) 165–184

[56] Kermarrec, A.M., Massoulié, L., Ganesh, A.J.: Probablistic reliable dissemination in large-scale systems. IEEE Transactions on Parallel and Distributed Systems **14**(3) (March 2003) 248–258

[57] Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC'06), New York, NY, USA, ACM (2006) 189–202

[58] Saroiu, S., Gummadi, P.K., Gribble, S.D.: Measuring and analyzing the characteristics of Napster and Gnutella hosts. Multimedia Systems Journal **9**(2) (August 2003) 170–184

[59] Eugster, P.T., Guerraoui, R., Kermarrec, A.M., Massoulié, L.: Epidemic information dissemination in distributed systems. IEEE Computer **37**(5) (May 2004) 60–67

[60] Jelasity, M., Kowalczyk, W., van Steen, M.: An approach to massively distributed aggregate computing on peer-to-peer networks. In: Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04), A Coruna, Spain, IEEE Computer Society (2004) 200–207

[61] Jelasity, M., Montresor, A., Babaoglu, O.: A modular paradigm for building self-organizing peer-to-peer applications. In Di Marzo Serugendo, G., Karageorgos, A., Rana, O.F., Zambonelli, F., eds.: Engineering Self-Organising Systems. Volume 2977 of Lecture Notes in Artificial Intelligence., Springer (2004) 265–282 invited paper.

[62] Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kermarrec, A.M., Kouznetsov, P.: Lightweight probabilistic broadcast. ACM Transactions on Computer Systems **21**(4) (2003) 341–374

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

[64] Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and its Applications. 2nd edn. Springer Verlag (1997)

[65] Marsaglia, G.: The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. Florida State University (1995) online at `http://www.stat.fsu.edu/pub/diehard`.

[66] Montresor, A., Jelasity, M.: Peersim: A scalable P2P simulator. In: Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing (P2P 2009), Seattle, Washington, USA, IEEE (September 2009) 99–100 extended abstract.

[67] Marsaglia, G., Tsang, W.W.: Some difficult-to-pass tests of randomness. Journal of Statistical Software **7**(3) (2002) 1–8

[68] Albert, R., Jeong, H., Barabási, A.L.: Error and attack tolerance of complex networks. Nature **406** (2000) 378–382

[69] Pastor-Satorras, R., Vespignani, A.: Epidemic dynamics and endemic states in complex networks. Physical Review E **63** (2001) 066117

[70] Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393** (1998) 440–442

[71] Newman, M.E.J.: Random graphs as models of networks. In Bornholdt, S., Schuster, H.G., eds.: Handbook of Graphs and Networks: From the Genome to the Internet. John Wiley, New York, NY (2002)

[72] DAS-2: http://www.cs.vu.nl/das2/

[73] Allavena, A., Demers, A., Hopcroft, J.E.: Correctness of a gossip based membership protocol. In: Proceedings of the 24th annual ACM symposium on principles of distributed computing (PODC'05), Las Vegas, Nevada, USA, ACM Press (2005)

[74] Barabási, A.L.: Linked: the new science of networks. Perseus, Cambridge, Mass. (2002)

[75] Dorogovtsev, S.N., Mendes, J.F.F.: Evolution of networks. Advances in Physics **51** (2002) 1079–1187

[76] Ganesh, A.J., Kermarrec, A.M., Massoulié, L.: Peer-to-peer membership management for gossip-based protocols. IEEE Transactions on Computers **52**(2) (February 2003)

[77] Law, C., Siu, K.Y.: Distributed construction of random expander networks. In: Proceedings of The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'2003), San Francisco, California, USA (April 2003)

[78] Pandurangan, G., Raghavan, P., Upfal, E.: Building low-diameter peer-to-peer networks. IEEE Journal on Selected Areas in Communications (JSAC) **21**(6) (August 2003) 995–1002

[79] Zhong, M., Shen, K., Seiferas, J.: Non-uniform random membership management in peer-to-peer networks. In: Proc. of the IEEE INFOCOM, Miami, FL (2005)

[80] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA, USA (February 2003)

[81] Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In Guerraoui, R., ed.: Middleware 2001. Volume 2218 of Lecture Notes in Computer Science., Springer-Verlag (2001) 329–350

[82] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), San Diego, CA, ACM, ACM Press (2001) 161–172

[83] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), San Diego, CA, ACM, ACM Press (2001) 149–160

[84] King, V., Saia, J.: Choosing a random peer. In: Proceedings of the 23rd annual ACM symposium on principles of distributed computing (PODC'04), ACM Press (2004) 125–130

[85] Kostić, D., Rodriguez, A., Albrecht, J., Bhirud, A., Vahdat, A.: Using random subsets to build scalable network services. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS 2003). (2003)

[86] Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In: Proceedings of ACM SIGCOMM 2003, ACM Press (2003) 395–406

[87] van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems **21**(2) (May 2003) 164–206

[88] Stavrou, A., Rubenstein, D., Sahu, S.: A Lightweight, Robust P2P System to Handle Flash Crowds. IEEE Journal on Selected Areas in Communications **22**(1) (January 2004) 6–17

[89] van Renesse, R.: The importance of aggregation. In Schiper, A., Shvartsman, A.A., Weatherspoon, H., Zhao, B.Y., eds.: Future Directions in Distributed Computing. Number 2584 in Lecture Notes in Computer Science, Springer (2003) 87–92

[90] Jelasity, M., Montresor, A., Babaoglu, O.: Detection and removal of malicious peers in gossip-based protocols. In: 2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability: Obstacles and Solutions (FuDiCo II: S.O.S.), Bertinoro, Italy (June 2004) invitation only workshop, proceedings online at `http://www.cs.utexas.edu/users/lorenzo/sos/`.

[91] Gupta, I., van Renesse, R., Birman, K.P.: Scalable fault-tolerant aggregation in large process groups. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN'01), Göteborg, Sweden, IEEE Computer Society Press (2001)

[92] Watts, D.J.: Small Worlds: The Dynamics of Networks Between Order and Randomness. Princeton University Press (1999)

[93] Ripeanu, M., Iamnitchi, A., Foster, I.: Mapping the gnutella network. IEEE Internet Computing **6**(1) (2002) 50–57

[94] Mitchell, T.M.: Machine Learning. McGraw-Hill (1997)

[95] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., Wawrzoniak, M.: Operating system support for planetary-scale services. In: Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI'04), USENIX (2004) 253–266

[96] Ghosh, B., Muthukrishnan, S.: Dynamic load balancing by random matchings. Journal of Computer and System Sciences **53**(3) (December 1996) 357–370

[97] Yalagandula, P., Dahlin, M.: A scalable distributed information management system. In: Proceedings of ACM SIGCOMM 2004, Portland, Oregon, USA, ACM Press (2004) 379–390

[98] Nekovee, M., Soppera, A., Burbridge, T.: An adaptive method for dynamic audience size estimation in multicast. In Stiller, B., Carle, G., Karsten, M., Reichl, P., eds.: Group Communications and Charges: Technology and Business Models. Number 2816 in Lecture Notes in Computer Science, Springer (2003) 23–33

[99] Horowitz, K., Malkhi, D.: Estimating network size from local information. Information Processing Letters **88**(5) (2003) 237–243

[100] Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM **27**(2) (1980) 228–234

[101] Dolev, D., Lynch, N., Pinter, S., Stark, E., Weihl, W.: Reaching approximate agreement in the presence of faults. Journal of the ACM **33**(3) (July 1986) 499–516

[102] Fekete, A.: Asynchronous approximate agreement. Information and Computation **115**(1) (November 1994) 95–124

[103] Madden, S., Szewczyk, R., Franklin, M.J., Culler, D.: Supporting aggregate queries over ad-hoc wireless sensor networks. In: Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02), Callicoon, New York, IEEE Computer Society Press (2002) 49–58

[104] Kutyłowski, M., Letkiewicz, D.: Computing average value in ad hoc networks. In Rovan, B., Vojtáš, P., eds.: Mathematical Foundations of Computer Science (MFCS'2003). Number 2747 in Lecture Notes in Computer Science, Springer (2003) 511–520

[105] Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project (1998)

[106] Sankaralingam, K., Sethumadhavan, S., Browne, J.C.: Distributed pagerank for p2p systems. In: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12 '03). (2003) 58–69

[107] Shi, S., Yu, J., Yang, G., Wang, D.: Distributed page ranking in structured p2p networks. In: Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03). (October 2003) 179–186

[108] Parreira, J.X., Donato, D., Michel, S., Weikum, G.: Efficient and decentralized PageRank approximation in a peer-to-peer web search network. In: Proceedings of the 32nd international conference on Very large data bases (VLDB'2006), VLDB Endowment (2006) 415–426

[109] Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The eigentrust algorithm for reputation management in p2p networks. In: Proceedings of the 12th international conference on World Wide Web (WWW'03), New York, NY, USA, ACM Press (2003) 640–651

[110] Koren, Y.: On spectral graph drawing. In: Proceedings of the 9th International Computing and Combinatorics Conference (COCOON'03). Number 2697 in Lecture Notes in Computer Science, Springer (2003) 496–508

[111] Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A decentralized network coordinate system. In: Proceedings of ACM SIGCOMM 2004, Portland, Oregon, USA, ACM Press (2004)

[112] Lubachevsky, B., Mitra, D.: A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit radius. Journal of the ACM **33**(1) (January 1986) 130–150

[113] Burgess, M., Canright, G., Engø-Monsen, K.: Importance-ranking functions derived from the eigenvectors of directed graphs. Technical Report DELIS-TR-0325, DELIS Project (2006)

[114] Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Reviews of Modern Physics **74**(1) (January 2002) 47–97

[115] Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H., eds.: Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide. SIAM, Philadelphia (2000)

[116] Albert, R., Jeong, H., Barabási, A.L.: Diameter of the world wide web. Nature **401** (1999) 130–131

[117] Frommer, A., Szyld, D.B.: On asynchronous iterations. Journal of Computational and Applied Mathematics **123**(1-2) (2000) 201–216

[118] Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04), Washington, DC, USA, IEEE Computer Society (2004) 4–10

[119] Sacha, J., Dowling, J., Cunningham, R., Meier, R.: Using aggregation for adaptive super-peer discovery on the gradient topology. In Keller, A., Martin-Flatin, J.P., eds.: Proceedings of the Second IEEE International Workshop on Self-Managed Networks, Systems and Services (SelfMan 2006). Volume 3996 of Lecture Notes in Computer Science., Springer (2006)

[120] Sacha, J., Napper, J., Stratan, C., Pierre, G.: Adam2: Reliable distribution estimation in decentralised environments. In: 2010 International Conference on Distributed Computing Systems, (ICDCS), IEEE Computer Society (2010) 697–707

[121] Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured peer-to-peer networks. In: Proceedings of the 16th ACM International Conference on Supercomputing (ICS'02). (2002)

[122] Adamic, L.A., Lukose, R.M., Puniyani, A.R., Huberman, B.A.: Search in power-law networks. Physical Review E **64** (2001) 046135

[123] Montresor, A.: A robust protocol for building superpeer overlay topologies. In: Proceedings of the 4th IEEE International Conference on Peer-to-Peer Computing (P2P'04), Zurich, Switzerland, IEEE Computer Society (August 2004) 202–209

[124] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making gnutella-like p2p systems scalable. In: Proceedings of ACM SIGCOMM 2003. (2003) 407–418

[125] Voulgaris, S., Kermarrec, A.M., Massoulié, L., van Steen, M.: Exploiting semantic proximity in peer-to-peer content searching. In: Proceedings of 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004). (2004) 238–243

[126] Voulgaris, S., van Steen, M.: An epidemic protocol for managing routing tables in very large peer-to-peer networks. In: Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, (DSOM 2003). Number 2867 in Lecture Notes in Computer Science, Springer (2003)

[127] Aberer, K., Datta, A., Hauswirth, M., Schmidt, R.: Indexing data-oriented overlay networks. In: Proceedings of 31st International Conference on Very Large Databases (VLDB), Trondheim, Norway, ACM (August 2005)

[128] Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology p2p systems. In: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), Konstanz, Germany, IEEE Computer Society (August 2005) 39–46

[129] Angluin, D., Aspnes, J., Chen, J., Wu, Y., Yin, Y.: Fast construction of overlay networks. In: Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). (July 2005) 145–154

[130] Voulgaris, S., van Steen, M.: Epidemic-style management of semantic overlays for content-based searching. In Cunha, J.C., Medeiros, P.D., eds.: Proceedings of Euro-Par. Number 3648 in Lecture Notes in Computer Science, Springer (2005) 1143–1152

[131] Massoulié, L., Kermarrec, A.M., Ganesh, A.J.: Network awareness and failure resilience in self-organising overlay networks. In: Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS 2003), Florence, Italy (2003) 47–55

[132] Bonnet, F., Kermarrec, A.M., Raynal, M.: Small-world networks: From theoretical bounds to practical systems. In: Principles of Distributed Systems. Volume 4878., Springer (2007) 372–385

[133] Patel, J.A., Gupta, I., Contractor, N.: JetStream: Achieving predictable gossip dissemination by leveraging social network principles. In: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications (NCA 2006), Cambridge, MA, USA (July 2006) 32–39

[134] Zhao, B.Y., Huang, L., Jeremy Stribling, A.D.J., Kubiatowicz, J.D.: Exploiting routing redundancy via structured peer-to-peer overlays. In: Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP 2003). (2003) 246–257

[135] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: OpenDHT: A public DHT service and its uses. In: Proceedings of ACM SIGCOMM 2005, ACM Press (2005) 73–84

[136] Koren, Y.: Embedder `http://www.research.att.com/~yehuda/index_programs.html`.

[137] Gummadi, K.P., Saroiu, S., Gribble, S.D.: King: Estimating latency between arbitrary internet end hosts. In: Internet Measurement Workshop (SIGCOMM IMW). (2002)

[138] Kalidindi, S., Zekauskas, M.J.: Surveyor: An infrastructure for Internet performance measurements. In: Proceedings of INET'99, San Jose, CA, USA (1999)

[139] Castro, M., Costa, M., Rowstron, A.: Performance and dependability of structured peer-to-peer overlays. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), IEEE Computer Society (2004)

[140] Foster, I., Kesselman, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers (1999)

[141] Saroiu, S., Gummadi, P.K., Gribble, S.D.: A measurement study of peer-to-peer file sharing systems. In: Proceedings of Multimedia Computing and Networking 2002 (MMCN'02), San Jose, CA (2002)

[142] Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks. In: Proceedings of the 10th ACM SIGOPS European Workshop. (2002)

[143] Foster, I., Iamnitchi, A.: On death, taxes, and the convergence of peer-to-peer and grid computing. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA, USA (February 2003)

[144] Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, MA (2001)

[145] Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley (April 2001)