UNIVERSITÉ DE NEUCHÂTEL

Load Management in Peer-to-Peer Systems: Structures and Algorithms

Thèse

présentée le 29 Mars 2010 à la Faculté des Sciences de l'Université de Neuchâtel pour l'obtention du grade de **docteure ès sciences**
par

Sabina Serbu

acceptée sur proposition du jury:

Prof. Peter Kropf	Université de Neuchâtel	directeur de thèse
Prof. Pascal Felber	Université de Neuchâtel	rapporteur
Prof. René Meier	Trinty College Dublin	rapporteur
Prof. Julien Bourgeois	Université de Franche Comté	rapporteur

Université de Neuchâtel

FACULTE DES SCIENCES
Secrétariat-Décanat de la faculté
- Rue Emile-Argand 11
- CP 158
- CH-2009 Neuchâtel

IMPRIMATUR POUR LA THESE

Load Management in Peer-to-Peer Systems : Stuctures and Algorithms

Sabina SERBU

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des Sciences de l'Université de Neuchâtel, sur le rapport des membres du jury

MM. P. Kropf (directeur de thèse), P. Felber, R. Meier (Trinity College Dubin, Irlande) et J. Bourgeois (Université de Franche-Comté, France)

autorise l'impression de la présente thèse.

Neuchâtel, le 20 avril 2010

Le doyen :
F. Kessler

UNIVERSITE DE NEUCHATEL
FACULTE DES SCIENCES
Secrétariat - décanat de la faculté
Rue Emile-Argand 11 - CP 158
CH-2009 Neuchâtel

Felix Kessler

■ Téléphone : +41 32 718 21 00 ■ E-mail : secretariat.sciences@unine.ch ■ www.unine.ch/sciences

To my family.

Acknowledgements

First, I would like to thank my adviser, Prof. Peter Kropf, for all his guiding, advice and understanding during my PhD studies. I would also like to thank Prof. Pascal Felber for our collaboration and his practical advice.

I have highly appreciated the great environment at our Computer Science Department. For this reason, I would like to thank all my colleagues for the academic discussions and for the nice moments we have spent together at the university and outdoors. A special thought for my office mate, Leo, a great person with a very good sense of humor who still owes me a tennis match.

As for the University of Neuchâtel, I have found it a very pleasant place for my studies. The SUN sports service allowed me to take advantage of a various choice of sports, while the French course given by dedicated professors helped me a lot especially in my first year spent in Switzerland.

Abstract

Keywords: Peer-to-Peer, load balancing, load reduction, information lookup, dissemination, gossiping, routing.

In this thesis, we present several novel techniques for load management in peer-to-peer systems. We tackle two types of peer-to-peer systems: information lookup systems (in particular DHTs), where we define new load balancing solutions, and information dissemination systems, where we define new methods for load reduction.

First, we introduce the context of peer-to-peer systems and we elaborate on the existing solutions on load management. We classify them into three different categories: object placement, traffic routing and underlay. While the first two categories are aimed mostly at information lookup systems, dealing with object-to-node assignment and the routing strategies to be applied for object lookup, the latter category is more general. Any overlay, regardless its applicability, needs to have some knowledge about its underlay in order to manage its traffic load.

Our contributions to load management solutions are threefold.

We propose HyPeer, a novel DHT overlay with namespace balancing that offers flexible-choice among several routing strategies. For this purpose, we have built the uniform and regular HyPeer structure, where the nodes are conscientiously placed in order to offer *path redundancy* at similar path lengths. Having multiple paths between any two nodes, many different routing strategies can be applied. We propose four routing strategies aiming the most common goals: short path length, low path delay, fault tolerance and, most important in our context, routing load balancing. They all achieve very good results at the cost of only few local computations to determine

the next hop in the request path. Moreover, the overlay offers support for defining new routing strategies or for refining the existing routing strategies with new metrics.

For other existing DHTs, we propose a routing balancing solution that can be applied to any overlay that allows flexibility in the choice of the neighbors. Our solution is adaptive and it is based on *link reorganization*: according to the load fluctuation in the system, the most loaded neighbors are discarded, the forwarding traffic being redirected to less loaded peers instead. This solution comes at low costs, no extra messages being involved and moreover triggering link reorganization only when the load reaches too high values.

In information dissemination systems, we propose a novel strategy in order to reduce the load at the underlay level. We do not use complete randomness as classical strategies do, this generating too much traffic load at the underlay, instead we consider *proximity awareness*. After a limited seeding of the network, we give preference to the usage of short routes for delivering the dissemination message. Our solution significantly reduces the traffic load at the underlay, while not affecting the dissemination time.

Résumé

Mots-clés: Pair-à-Pair, équilibrage de charge, réduction de charge, recherche d'information, dissémination, protocole épidémique, routage.

Dans cette thèse, nous présentons plusieurs techniques inédites pour la gestion de charge dans les systèmes pair-à-pair. Nous abordons deux types de systèmes pair-à-pair: les systèmes de recherche d'information (en particulier DHTs), où nous définissons de nouvelles solutions pour l'équilibrage de charge, et les systèmes de diffusion d'information, où nous définissons de nouvelles méthodes de réduction de charge.

Tout d'abord, nous présentons le contexte des systèmes pair-à-pair et nous élaborons sur les solutions existantes en matière de gestion de charge. Nous les classifions en trois différentes catégories: le placement des objets, le trafic de routage et la sous-couche (underlay). Bien que les deux premières catégories visent avant tout les systèmes de recherche d'information, portant sur l'affectation objet-à-nœud et sur les stratégies de routage appliquées pour la recherche d'objet, la dernière catégorie est plus générale. Tout recouvrement (overlay), quel que soit son applicabilité, doit avoir une certaine connaissance de son sous-couche afin de pouvoir gérer sa charge de trafic.

Nous apportons trois solutions à la gestion de charge dans les systèmes pair-à-pair.

Nous proposons HyPeer, un recouvrement de type DHT avec équilibrage dans l'espace de noms qui offre un choix flexible entre plusieurs stratégies de routage. À cette fin, nous avons construit la structure de HyPeer uniforme et régulière, où les nœuds sont consciencieusement placés. Le but est de fournir une *redondance des chemins* (path redundancy), où les chemins ont des longueurs similaires. Avec plusieurs chemins entre deux nœuds, de

nombreuses différentes stratégies de routage peuvent être appliquées. Nous proposons quatre stratégies visant les plus communs objectifs: chemin court, délai faible, tolérance de panne et, le plus important dans notre contexte, équilibrage de charge de routage. Nos stratégies atteignent toutes de très bons résultats au coût de juste quelques calculs locaux pour déterminer le saut suivant dans le chemin. En outre, le recouvrement soutient la définition de nouvelles stratégies de routage ou le raffinage des stratégies de routage existantes avec de nouvelles métriques.

Pour les autres DHTs existants, nous proposons une solution d'équilibrage de charge de routage qui peut être appliquée à tout recouvrement qui permet une flexibilité dans le choix des voisins. Notre solution est adaptative et elle est basée sur la *réorganisation des liens* (link reorganization): en fonction de la fluctuation de charge dans le système, les voisins les plus chargés sont écartés, le trafic étant dirigé vers des pairs moins chargés. Cette solution a peu de frais, ne génère pas de messages supplémentaires et la réorganisation des liens est déclenché que lorsque la charge atteint des valeurs trop élevées.

Dans les systèmes de diffusion d'information, nous proposons une nouvelle stratégie pour réduire la charge au niveau sous-couche. Nous n'utilisons pas le hasard comme les stratégies classiques le font, ce qui génère une charge de trafic très grande dans la sous-couche. Nous considérons plutôt la conscience de proximité. Après une dispersion limitée de l'information dans le réseau, nous donnons la préférence au routes courtes pour la livraison du message de diffusion. Notre solution permet de réduire considérablement la charge de trafic dans la sous-couche, tout en n'affectant pas le temps de diffusion.

Contents

List of Figures **xv**

List of Tables **xix**

Glossary **xxi**

1 Introduction **1**

1.1 Context 1

1.2 Motivation and Goals 4

1.3 Contributions 5

1.4 Evaluation Considerations 7

1.5 Organization of the Thesis 8

2 Background: Peer-to-Peer Systems **9**

2.1 Distributed Hash Tables (DHTs) 12

2.1.1 Chord 14

2.1.2 Pastry 15

2.1.3 CAN: Content-Addressable Network 18

2.2 Gossiping as a Means to Self-Organizing 20

2.2.1 Gossip-Based Protocols 21

2.2.2 Overlay Creation and Maintenance 23

2.2.3 Information Dissemination 25

2.2.4 Other Applications of Gossiping 26

2.3 Summary and Discussion 28

3 Load Balancing in Peer-to-Peer Systems 31
3.1 Introduction 31
3.2 Preliminaries on Load Balancing 33
3.2.1 DHT Overlay Guidelines 33
3.2.2 The problem: causes of imbalance 34
3.2.3 Classification and applicability of load balancing solutions 37
3.3 Object Placement 39
3.3.1 Namespace balancing 39
3.3.2 Virtual Servers 40
3.3.3 Using multiple functions to hash an object 44
3.3.4 Caching and Replication 45
3.4 Traffic Routing 50
3.4.1 Link Reorganization 51
3.4.2 Path Redundancy 53
3.4.2.1 Single routing strategy 53
3.4.2.2 Multiple routing strategies 55
3.5 Load Balancing in the Underlay 55
3.6 Summary and Discussion 59

4 HyPeer: Structured Overlay with Path Redundancy 63
4.1 Introduction 63
4.2 Background and Motivation 65
4.3 HyPeer Overlay 69
4.3.1 HyPeer Structure 69
4.3.2 HyPeer Routing Strategies 72
4.4 Evaluation 76
4.4.1 HyPeer Structure 77
4.4.1.1 Inter-node distance 77
4.4.1.2 Aligned neighbors 78
4.4.2 HyPeer Routing Strategies 80
4.4.2.1 Operation without failures 80
4.4.2.2 Operation with failures 84
4.4.3 Evaluation Review 86

4.5 Summary and Discussion 87

5 Adaptive Load Balancing by Link Reorganization 89

5.1 Introduction 89
5.2 Preliminaries 91
5.2.1 System Model 91
5.2.2 Implications of Zipf-like requests 91
5.3 Link Reorganization Solution 93
5.3.1 The LR Algorithm 94
5.3.2 Adding Caching as Complementary Solution (LR+C) 97
5.4 Evaluation 100
5.4.1 Evaluation of LR 100
5.4.2 Evaluation of LR+C 104
5.4.3 Zipf-like requests with different parameter 107
5.4.4 Evaluation Review 108
5.5 Summary and Discussion 109

6 Network-Friendly Gossiping 111

6.1 Introduction 111
6.2 Network Friendliness 114
6.2.1 Network-Friendliness through Underlay Proximity Awareness . . 114
6.2.1.1 Metrics for Proximity Awareness 114
6.2.1.2 Reducing measurement costs using estimations 116
6.2.2 Network-Friendly Gossip-based Dissemination 118
6.3 Evaluation 121
6.3.1 Topologies 122
6.3.2 Experimental setup 125
6.3.3 Time efficiency of the dissemination 126
6.3.4 Impact on the load at each router 127
6.3.5 Evaluation Review 129
6.4 Summary and Discussion 130

7 Conclusions **133**
7.1 Summary . . . 133
7.2 Contributions . . . 134
7.3 Discussion and Outlook . . . 136
Publications **139**
Bibliography **141**

List of Figures

2.1 Example of Chord with an identifier space of $2^{m=6} = 64$. 14

2.2 Pastry leaf set, routing table and neighborhood set of node 10233102 (figure from [71]). 17

2.3 Pastry overlay structure and routing principle. 18

2.4 Example of a CAN structure, with routing and joining samples (figure from [67]). 19

2.5 Example of shuffling with Cyclon. 23

2.6 Visualisation of a torus structure constructed with T-Man (figure from [36]). 25

3.1 Scenarios with imbalanced load. 36

3.2 Examples of usage of multiple hash functions. 45

3.3 Example of path replication. 48

3.4 Example of replica peer selection: for request load distribution, the peers with higher capacity receive more requests. 50

3.5 Flexibility in the choice of neighbors. 52

3.6 Example of path convergence towards the same destination, node n_{24}. . . 53

3.7 Load balancing on the forwarding traffic towards the same destination, node n_{24}, using random choice for the next hop. 54

3.8 Request paths from A to D, with and without proximity routing. 57

4.1 Representation of HyPeer as an approximation of a hypercube. 66

4.2 Several different paths can be used to cover a given distance (140 here) between the source and the destination nodes. 66

4.3 The request path to be followed by two routing strategies (filled lines) from source node n_2 to destination node n_{24}. The incoming links of node n_{24} are marked with dotted lines. 67
4.4 Traffic on incoming links of node n_{24}. 68
4.5 Aligned neighbors of node n_{16} when a new node joins the overlay. 71
4.6 Routing path from node n_2 to node n_{24} for each routing strategy. 75
4.7 Percentages of nodes with an inter-node distance of 2^i. 77
4.8 Percentages of nodes with n aligned neighbors. 78
4.9 Percentages of nodes with an aligned neighbor at entry e. 78
4.10 CDF of routing path length. 82
4.11 Average number of used (outgoing) links per node, under different load-flows. 83
4.12 Routing load on the most loaded link per node, under popular requests (uz load-flow). 83
4.13 Percentages of failed requests under failures. 85
4.14 Average path length under failures. 86

5.1 Statistics of received and forwarded requests. 92
5.2 Request and routing load. 93
5.3 Routing tables before reorganization for load balancing. 94
5.4 Routing tables after reorganization for load balancing. 95
5.5 Example of functionality of the caching method. 100
5.6 Load distribution without load balancing (run0). 101
5.7 Load distribution with load balancing (run2). 101
5.8 Evolution of the number of updates over time (100 requests per unit of time) in the first 200 time units (run2) 102
5.9 The 300 most loaded nodes, without load balancing (run0). 103
5.10 The 300 most loaded nodes, with load balancing (run2). 103
5.11 Load distribution for the 300 most loaded nodes, using a leaf set of 8 nodes (run2). 104
5.12 Load balancing using dynamic routing table updates (LR run1). 105
5.13 Load balancing using dynamic routing table updates (LR run2). 105

5.14 Load balancing using dynamic routing table updates and caching (LR+C run1). 106
5.15 Load balancing using dynamic routing table updates and caching (LR+C run2). 106
5.16 LR and LR+C using Zipf's $\alpha = 0.5$ (run2). 108
5.17 LR and LR+C using Zipf's $\alpha = 2$ (run2). 108

6.1 Push-only dissemination: coverage (ratio of peers notified), complete (ratio of dissemination that notify *all* peers) and redundancy ratio for various f and *htl*. 119
6.2 Peer selection strategies. Dotted circles represent close views. Solid lines correspond to pushes and dashed lines to pulls. 121
6.3 Synthetic topologies. 122
6.4 Characteristics of the five topologies. 123
6.5 Distribution of the number of cycles required for receiving the *first* message. 126
6.6 Distribution of load on all routers: route lengths. 127
6.7 Distribution of load on all routers: bandwidth. 128

List of Tables

1.1 Our contributions per load balancing context. 5

2.1 Gossip usage. 22

3.1 Causes of load imbalance. 35
3.2 Load balancing solutions. 38
3.3 Comparison between load balancing solutions that use virtual servers. . 43

4.1 Routing Strategies in HyPeer. 72
4.2 Load-flows: selection of initiator nodes and requested keys. 77
4.3 Routing path length statistics. 81
4.4 Routing path length statistics in a FH. 81
4.5 Average request delay. 84

5.1 Load Statistics (average and variance) for all runs, using a leaf set of 4 and 8 nodes, respectively. 104
5.2 Load Statistics. 107
5.3 Load Statistics using $\alpha = 0.5$ and $\alpha = 2.0$ Zipf parameter. 108

Glossary

CC	Push Close, Pull Close
CR	Push Close, Pull Random
RC	Push Random, Pull Close
RR	Push Random, Pull Random
LR	Link Reorganization
LR+C	Link Reorganization and Caching
RTT	Round-Trip Time
htl	Hops To Live
TTL	Time To Live
AS	Autonomous System
BGP	Border Gateway Protocol
DHT	Distributed Hash Table
ID	Identifier
ISP	Internet Server Provider
NFGossip	Network-Friendly Gossip
P2P	Peer-to-Peer

Chapter 1

Introduction

1.1 Context

There has been a major paradigm shift in distributed computing over the last decade, with the emergence of many large-scale and widely distributed applications based on the peer-to-peer (P2P) paradigm. Examples include BitTorrent or Skype, which provide file-sharing capabilities, audio and video conferencing, message transfers, etc. Their functionality relies on *overlay networks* that connect participating peers via application-level communication links.

Overlay networks represent an interface to the underlying network for all participating peers, offering a friendly way for communication. They allow the communication between a source and a destination peer through a network (usually the Internet, but henceforth called the "underlay network") without the need of knowing the identity of the destination at the underlying level (for Internet, the IP address). Moreover, in the overlay, messages are routed from peer to peer, until the destination is reached, in a transparent manner to the end-users. The communication may serve for any purpose, such as spreading information or inquiring the system for some information. Peers send messages to other peers, containing either requests or information. The messages are then received by other peers which can forward them further on or reply to them. As a consequence, peers act as both servers and clients.

While overlay networks aim to simplify communication between peers, the complexity that is required to offer a reliable service is kept inside the overlay management system. We name a few issues that overlays usually deal with. The overlay needs to

remain connected in order to reach all peers, so it has to assure connectivity. It may deal also with certain security issues, such as the detection of malicious peers. It has to be fault tolerant in order to deal with peer and link failures. Also, it has to be scalable in order to allow the arrival of new peers and to treat peer departures. More interesting to us, peer-to-peer systems have to keep the idea of fairness, in order for each participating peer to take part with the same effort (i.e., volume of tasks per capacity) as any other peer, and thus not to overload some resources (peers or links). This effort is measured through the *load*. Ideally, the load will be proportional to the capacity of the individual peers. This is where we focus our research.

Being decentralized, peer-to-peer systems do not have a central unit to strictly control the load of each peer or link. A highly loaded peer may fail to respond, while loading too much a link may create bottlenecks and moreover affect the performance of other applications running on the same network. To avoid such problems, specialized distributed mechanisms are required for managing the load, for providing fairness and for properly exploiting the underlying network, but most importantly, to assure that messages arrive with no significant delays.

The high peaks of load that may occur for resources can be moderated through two solutions that deal with the system load. The first is to *reduce as much as possible the system load*. This is of course limited by external factors such as the quantity of messages that are issued in the system, which obviously do not depend on the overlay management. The second solution (that can be also a complement to the first solution) is to *share the system load* between more resources in order to reduce the differences in load between them. This latter solution is called load balancing.

Dealing with load is a rather wide issue, since *load* itself can represent different things. When a system is not performing efficiently and the cause is related to the load (i.e., too much load and/or uneven share), one has first to answer to the question: *What is the load problem?* Since both peers and links can be loaded, the possible reasons vary from the quantity of information to routing issues. Most of the causes come from either an inappropriate infrastructure for the application or the way it is used. Once the load problem is recognized, it needs to be analyzed. Metrics are put into place in order to evaluate the current "value" of the load. When these numbers show exactly where the problem is and how severe it is, the question is *How to reduce this load?* or *How to balance it?* In the former case, one needs to find ways to reduce the load

that is generated by the input of the system (e.g., the messages of the users), while in the latter case, one needs to identify the available resources to share this load, which extends the question by *Who will participate?*

We give two simple examples to depict this analysis and possible solutions. First, take an overlay that is offering capabilities for the lookup of information or content, such as for example in a sharing system. Here, each peer owns some information (files). Consider a peer that owns much more files compared to the other peers of the system. This peer is clearly disfavored, since not only it needs to allocate more storage space, but it also has to serve more requests than the other peers. The load problem is the high number of files at this peer. If the load is reduced, i.e., files are deleted, they are lost. This is a loss for all the overlay, since these files may not be found elsewhere, and thus this solution cannot be taken into consideration. Alternatively, if the files would be shared with other peers, such as peers in the neighborhood of the loaded peer, its load would be significantly decreased, while slightly increasing the load of the neighbor peers.

Consider now as a second example an overlay that is used for spreading information, e.g., a news service. Each peer is issuing news that are sent to other peers in a random manner. At some time, an important information needs to be sent to all peers, in which case all peers collaborate in propagating the news. This process can easily lead to bottlenecks due to the high number of messages that need to be transmitted in the system. The problem is definitely the large number of messages and the choice of paths they are taking. To solve it, more intelligent alternatives for the message spreading algorithms (such as minimization of useless messages that are sent to peers that have already received the news and the choice of paths for useful messages) can balance the load, and, more adequate in this case, can significantly reduce it.

Through our examples we also show that the most common load problems come from the quantity of messages and/or the paths they follow. In this thesis, we manage the load focusing on both of these aspects, with a larger bias on the latter, i.e., the *routing* support. Routing is related to the choice of the set of neighbors for a peer and the choice of one of these neighbors to forward a message. We will thus cope with flexibility to offer multiple choices: where flexibility exists, we will benefit from it; and where it does not, we will design it.

1.2 Motivation and Goals

We have analyzed two types of systems: information lookup and information dissemination systems.

A common characteristic of information lookup systems is the greediness of its requests: each message tries to use the best resources of the overlay while largely ignoring the other messages. Usually, a request aims to use short communication paths. A peer sends a message to another peer from the system and it expects to reach it in a short number of steps or hops, in an attempt to obtain a short response time. Another example is the choice of a small set of peers to be used as intermediary hops for a large part of the communication because they have been the longest in the system and thus are considered to be more reliable. These greedy goals may provide good results under ideal conditions (such as a low usage of the overlay), however selfishness, even intuitively, is not a good approach in achieving efficiency in distributed systems. System parameters and the existing communication in the overlay are ignored and some resources are prone to become more loaded than others. The overlay has the load hardly balanced between participating peers.

Another issue concerns the correspondence between the overlay and the underlay network. A message that follows a straight path in the overlay structure might follow an inefficient path in the underlay network, traversing more routers than needed, which would obviously charge more underlying resources. Again, this might not be a problem in applications that send only few messages, however, in systems where the communication involves a high volume of messages, this might cause severe effects. This is the case in information lookup systems that have many requests for popular items, and moreover in information dissemination systems.

All these problems can be attacked through optimizations, however these solutions might be limited by the infrastructure itself, such as not allowing enough flexibility, which is essential for sharing the load. Consequently, for good performance results in load balancing and load reduction, load management has to be thought of from the design time of an overlay, in order to avoid later load problems.

It is very important to analyze the repercussion and the trade-offs of a load balancing solution, in order not to affect in a negative way the fault tolerance or other important properties of the overlay. It is thus challenging to determine which resources should

be chosen to share the load. Moreover, these resources might balance well the load, but after some time (i.e., after some peers have joined or left the system, or an object becomes suddenly popular), other resources might better be used instead. Therefore, it is mandatory for any solution to be adaptive to overlay changes.

Our goals are to provide efficient adaptive mechanisms to balance the load in the overlay and to reduce it in the underlay. We will be tackling the overlay level, achieving our goals through collaboration between peers and flexibility in the overlay design and, where appropriate, through input from the underlay.

1.3 Contributions

The context of our contributions is twofold. We tackle *load balancing* in information lookup systems, more specifically in the area of Distributed Hash Tables (DHTs). As we will further detail in Chapter 2, DHTs use hash functions for mapping (assigning) objects to peers for efficient lookup. We will be treating different load issues, mostly focusing on routing. Additionally, we tackle *load reduction* in information dissemination systems, as communication-intensive overlays that are prone to overload. Our solutions are based on self-organization, also detailed in Chapter 2. In self-organizing behaviors, peers collaborate in order to achieve a common emergent goal.

Through our solutions, we manage the load at different levels: namespace balancing, caching and replication, traffic routing and underlying topology. All these notions, together with the related work in the field are described in Chapter 3.

Load Balancing Context	**Our Solution**
Namespace Balancing	HyPeer
Caching and Replication	Adaptive Load Balancing by Link Reorganization
Traffic Routing	Adaptive Load Balancing by Link Reorganization, HyPeer
Underlay	Network-Friendly Gossiping

Table 1.1: Our contributions per load balancing context.

Table 1.1 shows our contribution per load balancing context. We provide the following solutions: (1) HyPeer, a structured overlay with path redundancy, (2) adaptive load balancing by link reorganization (which has also a complementary solution of caching and replication) and (3) network-friendly gossiping.

HyPeer. HyPeer is our concept of DHT overlay that is meant for achieving different goals for any lookup request, favoring user needs (such as short response time) or overlay management (such as load balancing or fault tolerance). Each goal is achieved through a different routing strategy and lookup requests can be forwarded using any of them. The choice of the routing strategy to use can be done either at the initiator peer (especially for greedy reasons) or by any peer on the path (e.g., considering current overlay conditions). We have designed HyPeer to offer routing strategies for balancing the traffic load, but also to achieve fault tolerance, short delays and short paths. Additionally, HyPeer allows support for other routing strategies. The idea behind it is *path redundancy*: the overlay is able to route along several paths between any two peers, and the choice for the next peer in the routing path is given by the routing strategy.

In HyPeer we provide load balancing in two ways, through: (a) the overlay construction with *namespace balancing*, by a joining mechanism that uniformly places the peers in a hypercube-like structure and (b) the routing strategy for *balancing the routing traffic*. To the latter, we can also add the intuitive balance of the routing traffic when the routing strategies are random-uniformly used. Moreover, the load at the underlay level can be reduced when using a proximity routing strategy.

Adaptive Load Balancing by Link Reorganization. Our second contribution is based on *link reorganization*, and it is also meant for balancing the routing traffic in DHTs, but in a different manner. Here, we deal with the overlay links and not the routing strategy: the links change the peers that they point to, according to load measurements, in order to reduce the traffic passing through loaded peers. This traffic is forwarded to less loaded peers instead, balancing the routing load in the system. We show the efficiency of this solution in a DHT that uses prefix routing, while running lookup requests. The load information travels with the lookup requests, which offers high adaptivity to load changes.

The simplicity of our solution makes it applicable to any overlay that allows flexibility in the choice of the neighbors. Moreover, it can be complemented by a *caching* strategy in order to balance also the request load. For improved efficiency, the caching strategy also makes use of the load information of the peers (when deciding whether a peer is globally loaded or whether an object should be replicated).

Network-Friendly Gossiping. To cover communication-intensive applications, we tackle gossip-based dissemination where we reduce the amount of traffic that travels in the underlay network.

The two known models for gossip-based dissemination are *push* and *pull*, which in our solution, are used in a non-classical manner: first we seed the network with the epidemic message during a limited push phase that would not generate too many duplicates, then we let dissemination complete through a pull phase. We use *network-friendly* protocols that are based on *proximity awareness* in order to select the partner peer in each gossip exchange. We elaborate on three possibilities and we compare them with classical gossip-based dissemination that ignores the underlying topology. We analyze their effects on several topologies, both synthetic and real.

1.4 Evaluation Considerations

We have performed our evaluations mostly by means of simulations. For our first two contributions, both in the context of information lookup in DHTs, we have used our own simulator implemented in Java. We have implemented both DHT structure creation and routing strategies, while complying to the guidelines for a common API proposed in [15].

We have considered churn only in HyPeer. Various experiments have been run under different churn scenarios in order to analyze how churn affects the structure. Moreover, we have evaluated our routing strategies under different node failure rates. We have not considered churn in our link reorganization solution and neither in the network-friendly gossiping since it was not expected that churn affects the load management significantly. Both these solutions deal with updating the neighbors according to some metrics, while always taking into consideration live nodes.

For the simplicity of the presentation of our solutions, we have assumed homogeneity (same characteristics for all nodes) and not considered concurrency issues. We do not expect these limitations to significantly reduce the efficiency of our solutions.

For information dissemination with network-friendly gossiping, we have used the Rappel simulator [62] implemented in C++, which is based on traces of Internet topology, latency measurements, churn and RSS subscriptions. We have performed our

evaluations on different synthetic and real topologies, where, in order to better simulate network dynamics, we have also considered small variations in the delays of each topology. We have also performed experiments on PlanetLab [63], this time to evaluate our means for proximity measurement.

1.5 Organization of the Thesis

The thesis is organized as follows. In Chapter 2 we first present a description of peer-to-peer systems, together with their classification and characteristics, while giving examples of the most relevant existing overlays to our research. We then describe self-organization and give an overview of gossiping as means of self-organizing behaviors. In Chapter 3 we discuss the state of the art in load balancing, in a survey where we group the existing solutions by different meanings of load. We then proceed with elaborating on our solutions for load balancing. We present our two solutions for balancing the routing traffic in Chapter 4, where we detail HyPeer, our overlay with path redundancy for flexible-choice routing, and in Chapter 5, where we present our link reorganization solution that can be complemented by a caching mechanism. Our network-friendly gossip solution for reducing the traffic at the underlying level during information dissemination is presented in Chapter 6. We summarize, conclude and give an outlook on future work in Chapter 7.

Chapter 2

Background: Peer-to-Peer Systems

"Build for your team a feeling of oneness, of dependence on one another and of strength to be derived by unity."

Vince Lombardi

In this chapter we give a background on peer-to-peer systems and we present the most relevant work in the field, as context for our research in load reduction and balancing.

A peer-to-peer system is a distributed system where peers (computers) communicate in a decentralized way. Peer-to-peer systems are mostly used for information transmission, such as file sharing, content distribution, video/audio transmission. During the last decade, several peer-to-peer systems have been widely employed. The most known are KaZaA, Skype, Napster, BitTorrent, Gnutella, etc. These systems are built on top of a network (or another system) that is used as support for communication.

Peer-to-peer systems appeared from the need of decentralization. There is no central entity to act as application server, or to coordinate or perform management tasks. The advantages are the removal of the single point of failure (i.e., the system continues to perform even if a resource fails, using other resources instead) and the distribution of the tasks between the peers. Each peer acts both as a server and a client, serving and being served by the system, respectively. To enable communication, peers collaborate in a self-organizing manner.

Peer-to-peer and grid systems are both decentralized systems running on top of a network, however their objectives are different. Peer-to-peer systems focus on communication between peers that is needed for a certain application. In contrast, grid systems focus on tasks that peers are supposed to perform and on finding the proper peers for particular tasks.

A peer-to-peer system connects participating peers in an *overlay*. Peers are connected through *logical links* (or simply called *links*) that represent routing paths in the underlying network. The most used underlay network is the Internet or IP networks in general. In order to communicate, each peer knows a subset of the participating peers, which are called *neighbors*. The overlays may differ in their infrastructure (e.g., restricted/unrestricted number of neighbors), management (e.g., dedicated peers or completely decentralized) and purpose (i.e., the application for which the overlay was built for).

There are three communication operations that are application-independent and that any peer needs to know how to perform: receive a message from another peer, send a message to another peer and forward a message. Besides them, there are application-specific operations, such as finding the right neighbor to send/forward a message to, initiate some requests in the overlay, reply to incoming requests.

Additionally, two other operations define the life time of a participating peer. The process of becoming part of an overlay is called *joining*. Depending on the overlay, in order to connect to other participating peers, a new peer p may receive a list of other peers (called bootstrap peers) or it has to find some peers by itself. The peer p becomes thus part of the overlay by connecting to a set of peers, through logical links. However, this is not enough. Peer p needs to be known also by the other peers. For this reason, some overlays require additional messages, while others rely on the communication in the overlay in order to gradually create these links. The life time of a participating peer ends with its *departure*. A peer can either fail (i.e., silently leave) or it can announce its departure. Different methods are applied in each case and some overlays put in place mechanisms for fault tolerance in order to support these departures. In the case of failures, the overlay has to reorganize (i.e., recreate links) over time. A neighbor is detected as failed when it fails to receive a message or during periodical checks. Then, it is not considered a neighbor anymore. For overlays that need a specific number of neighbors, other suitable peers need to be found.

The rate of the arrivals and departures is referred to as *churn*. Often, the performace of a newly proposed overlay is analyzed and evaluated under churn. Some overlay protocols, such as maintenance or even load balancing solutions, might themselves require peers to leave and then rejoin the overlay, as in [27].

A message might be sent to one or several peers, depending on the application. Messages do not always reach a certain destination or they might arrive with delays. This is mostly the case when peers or links fail or, in some cases, when the messages have traversed too many other peers.

The creation and maintenance of the logical links determines the class of the peer-to-peer system. A common nomenclature differentiates between unstructured and structured peer-to-peer systems. A system without any constraints on its logical links, having usually arbitrary links between the peers, is commonly called *unstructured system*. Gnutella [25] (pre v0.4) and Freenet [21] are examples of unstructured systems. The management of this kind of systems is rather basic. A peer keeps some links towards some other peers and requests are sent without any sense of direction using partial or complete flooding (i.e., sent to a subset or to all neighbors). This method not only charges the system with a high traffic, it also does not assure that any message will reach its destination. In contrast, a *structured system* assures a high reachability of the destination peer and does not generate as much traffic. However, their management is more complex. Peers have identifiers and they maintain rather strict connections between them: specific constraints common for all peers are imposed on the neighbors, e.g., neighbors having to match certain patterns for their identifiers. Additionally, they define specific rules for message transmission, as we will discuss in the following section.

The mostly spread branch of structured peer-to-peer systems are unquestionably the **Distributed Hash Tables (DHTs)**. A DHT is a system where objects are assigned to different peers, based on a hash function, as follows. Additionally to peers having identifiers, each object (information) also has an identifier, which is obtained by applying a hash function on some attribute of the object, such as its name. The objects are distributed on the peers in the system according to their identifiers, e.g., an object is assigned to the peer with the closest identifier to the object's identifier. The main purpose of DHTs is *object lookup*: a peer issues a request for an object and the request is forwarded in the system until the destination peer is found (or some failure occurs), The forwarding process gives a sense of direction: each peer in the path chooses the

following peer based on its neighbors identifiers and the object identifier. When the destination peer is found, it can directly communicate with the inquiring peer. These systems are interesting to analyze from the load balancing point of view, since details like the choice of identifiers for peers or objects, the association of the objects to the peers, the forwarding process at the overlay, the generated traffic at the underlay level, etc., can easily generate a load imbalance that needs to be solved. Peer participation in the overlay tasks is considered alike for all, thus whatever the load is, it also needs to be equally distributed over the peers and links. This is where we first aim our attention at.

As decentralized systems, all peer-to-peer systems rely on communication in order to accomplish their tasks, either for maintenance or for the end-user inquiries. The ability of the overlay to act in a distributed, decentralized manner by having all peers collaborate producing emergent behavior is called **self-organization**. The goals can be very diverse: to achieve some kind of synchronization between the peers, to make computations based on distributed data (e.g., aggregate them), even to build an overlay (such as a DHT) by choosing the neighbors accordingly, etc. We will be focusing on gossiping protocols as means for self-organization. As this type of communication can generate a high amount of load, our second target lays in this area.

In the following sections of this chapter we first present DHTs, describing the most well-known systems. We then present self-organizing networks and elaborate on gossip-based protocols.

2.1 Distributed Hash Tables (DHTs)

A DHT is a peer-to-peer system that uses a hash function to distribute objects (information) over the peers. DHTs are mostly used for information lookup: a peer inquires the system for some information to which other peer(s) from the system replies. The request travels in the system from peer to peer.

In the past decade, several DHTs have been proposed. Basically, these DHT approaches differ in the hash space they consider, the rules for associating objects to peers and the routing strategies. We discuss them in the rest of this section.

The hash space (henceforth called identifier space) can be represented as a unidimensional or multidimensional space (2D, 3D, etc.). It may be a ring, Euclidean space,

hypercube, or any other type of graph. In this space, peers are less formally called *nodes*, while objects are referred to as *keys*. The objects are assigned to peers based on peer and object identifiers. Usually, a key is mapped to the closest or to the following node in the identifier space (according to a predefined order), but other methods can be employed as well. The node identifiers and their logical links represent the structure of the overlay, thus the identifier space has to be chosen accordingly.

Since DHTs deal with node failures, we use the following terminology. A node is *live* (or *alive*) if it actively participates in the lookup protocol, i.e., it can reply and forward requests. Conversely, a node is *dead* (or, it *failed*) if it cannot be contacted anymore and, as a consequence, it cannot be used to forward requests.

Each peer may issue a request for an object. The process of forwarding the request from peer to peer, from source until destination, is called *routing*. Each peer keeps its neighbors in a *routing table*, which has a fixed number of entries. Thus, when a peer fails, it has to be replaced by a new peer. Usually, there are strict rules for the peers at specific entries (as we will see in the DHT examples the following subsections), thus, finding another suitable peer, when it exists, requires additional messages. These operations represent the maintenance costs of the routing tables. The choice of the neighbor to forward a request to is given by the *routing strategy*. DHTs have various routing strategies, that depend on the overlay structure.

One notable difference between the DHT structures lies in the node *degree*, i.e., in the number of neighbors with which a node maintains continuous contact for supporting the routing mechanism. A *constant node degree*, which is usually a small number, assures low maintenance costs for operations such as maintaining routing tables or exchanging control information (e.g., state of neighbors). Unfortunately, this also means that this type of overlays do not offer a significant number of alternative paths (not even in failure-free cases). Examples of such designs include de Bruijn-based overlays [41], Viceroy [57] or CAN [67].

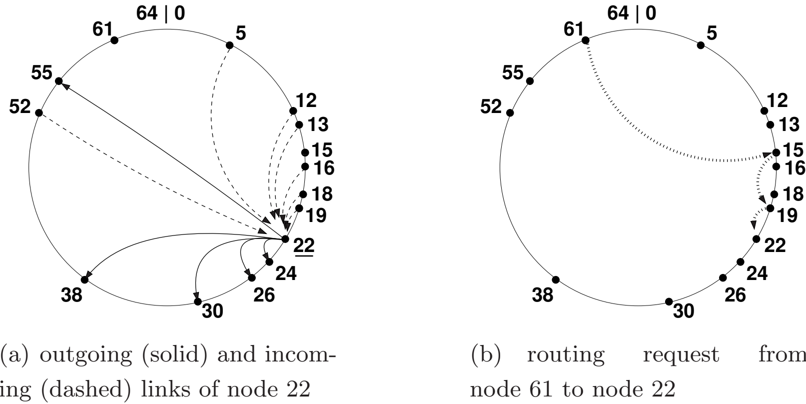
Other DHTs use a *logarithmic node degree*. Examples of such structures include Chord [85], Pastry [71], Tapestry [97] or Kademlia [59]. While these systems induce higher costs for maintaining the multiple entries in the routing tables, they allow for defining routing strategies that exploit alternative paths, using alternative routing table entries for routing a request. For instance, such paths can be used in case of a node failure, as in [78]. Alternative paths have not only the advantage of providing better

fault tolerance, but they also offer support for multiple routing strategies. For this reason, in our research, we are focusing on this type of overlays. With this support for alternative paths, we focus on the most common issues that overlays cope with and define corresponding routing strategies in Chapter 4. Moreover, some of these overlays allow for flexibility in the choice of the neighbors, which we will exploit later in Chapter 5.

As DHTs, we present Chord [85] and Pastry [71], which are the most typical examples of systems with a ring structure, that employ greedy-routing (more specifically prefix-routing for Pastry), and CAN [67] as typical example of overlay with a multidimensional identifier space.

2.1.1 Chord

In Chord [85], each node and key has a m-bit identifier on a 2^m identifier space designed as a ring, obtained by respectively hashing the IP address and the name. Each key is mapped to the first node (starting at the key identifier) that follows clockwise on the ring. For routing purposes, each node has a routing table with m entries, each entry i pointing towards the first node on the ring at a distance of at least 2^i, where $i = 0..m-1$. The node at entry i is also called *finger i*. The corresponding links are referred to as *incoming links* on the node they point to.



(a) outgoing (solid) and incoming (dashed) links of node 22

(b) routing request from node 61 to node 22

Figure 2.1: Example of Chord with an identifier space of $2^{m=6} = 64$.

A graphical representation of a Chord ring example is shown in Figure 2.1, on an identifier space of 2^6 addresses with 15 nodes. Figure 2.1(a) shows the outgoing and

incoming links of node 22, with solid and dashed lines, respectively. Fingers do not point to nodes at a distance exactly equal to a power of 2. The same applies also to the incoming links: they do not come always from distances equal to a power of 2 (for example, node 5 comes from a distance of 2^4+1). Chord uses *greedy routing*, a routing strategy that sends the requests as close as possible to the destination. The average path length is in the order of $O(\log N)$. The path is clockwise on the ring, as in the example from Figure 2.1(b): $61 \rightarrow 15 \rightarrow 19 \rightarrow 22$.

In order to assure connectivity and to facilitate the join and leave mechanisms, each node also keeps track of its predecessor. Note that any node knows its successor, which is its first finger. When a new node n_j joins the system, links are created/updated and the responsibility of the keys is reconsidered, as follows. Node n_j creates links towards its predecessor and successor and requests them to consider n_j as their new successor and predecessor, respectively. Node n_j builds its own routing table and the other nodes from the system update their routing tables in order to reflect its arrival. With the links being created, the node n_j obtains from its successor the objects for which it is now responsible for.

A stabilisation mechanism is used for the nodes that fail or leave voluntarily: periodically, each node n_s checks its successor, whether it is still alive, the right node and it has n_s as its predecessor. Moreover, and also periodically, each node n_s refreshes its routing table, by finding the right nodes for each entry. A node that leaves voluntarily gives the responsibility of its objects to its successor, in order to preserve the rule of key mapping to nodes.

To deal with failures, each node maintains a successor-list of r nodes (i.e., its r nearest successors on the ring). Whenever a request needs to be sent to a finger that is not reachable anymore, a lower finger is used instead and, if necessary, the nodes in the successor-list can be also used as alternatives. Objects can also be replicated at several successors. Chord has thus the advantage of simplicity and has been proved [85] to scale well with the number of nodes and to recover after high rates of churn.

2.1.2 Pastry

Like Chord, Pastry [71] has also a ring structure, where a node identifier is the result of hashing its IP address or its public key, and the key of an object is determined by hashing its name.

In Pastry, each node and key has an identifier with a sequence of digits in base 2^b (b has a typical value of 4) that determines its position on the ring. Each node is responsible for the closest keys (towards its predecessor and successor) on the ring.

Pastry routes requests to the node that is numerically closest to the destination key. Routing takes less than $\lceil log_{2^b} N \rceil$ steps, in a network of N nodes. At each step, the request is sent to a neighbor that shares with the key a prefix that is at least one digit longer than the common prefix of the key with the local node. If no such neighbor exists, the request is sent to a neighbor that shares exactly the same prefix as the local node but is numerically closer to the key. In order to support this routing procedure, each Pastry node maintains a *routing table* and a *leaf set*. When receiving a request for an object O, the request is forwarded to the node from the routing table or from the leaf set whose identifier has the longest common prefix with O.

The routing table is composed of $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. The ith entry in the routing table of node n_j maps to $2^b - 1$ nodes that share a common prefix of exactly i digits with node n_j. If no node is found suitable for an entry, that entry is left empty. However, this is unlikely to happen due to the uniform distribution of the nodes in the identifier space. The leaf set (denoted L) contains the numerically closest $|L|$ neighbors on the ring: the numerically closest $|L|/2$ nodes among its successors and the numerically closest $|L|/2$ nodes among its predecessors.

Besides the routing table and the leaf set, a Pastry node also has a neighborhood set M, which contains nodes that are closest, according to a proximity metric, to the local node. The neighborhood set is mostly used to maintain locality properties and it is used for routing only when neither the routing table nor the leaf set contain a node that can forward the request. The metric is usually the number of IP routing hops or the geographic distance between two nodes. Typical values for $|L|$ and $|M|$ are 2^b or 2×2^b.

When joining the system, a node creates its routing table as follows. A newly arrived node X sends a request towards its own identifier through a known node A (such as a node that is close according to the proximity metric). Node A is considered not to share any prefix with X, thus the first row in the routing table of A would perfectly fit as the first row in the routing table of X. All the nodes on the path send part of their routing tables to X: the first node B in the path (after A) shares a prefix of length 1

with X, and so B's first row is appropriate for X's first row, and so on with the other nodes on the path.

NodeId 10233102

Leaf set (SMALLER | LARGER)

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 2.2: Pastry leaf set, routing table and neighborhood set of node 10233102 (figure from [71]).

An example of routing table is shown in Figure 2.2. In a Pastry system where all nodes have an identifier as a sequence of digits in base $2^{b=2} = 4$, the figure shows the leaf set, routing table and neighborhood set of node 10233102. The leaf set contains $|L|$=8 nodes, half with smaller and the other half with larger identifiers than 10233102. The routing table has 8 rows, starting with row 0. Each row i contains 3 nodes, each one with a different digit after the common prefix of length i with 10233102. Row 0 contains nodes that do not share any prefix with 10233102, so their first digit is either 0, 2 or 3 (different from 1, which is the first digit of 10233102). The nodes at row 1 share the first digit with the local node so their first two digits are 11, 12 and 13, and so on with the rest of the rows. The shaded cells do not contain any node, they only show the corresponding digit of the local node. The neighborhood set contains $|M|$=8 nodes that have been chosen according to a predefined proximity metric. The corresponding IP addresses are not shown.

Figure 2.3 shows another example of Pastry overlay, this time for routing, where node N_4 sends a lookup request for key K_{24}. The Pastry parameters are b=1, $|L|$=2 and identifiers on 5 digits. From N_4, the request is sent to node N_{28} (the node numerically closest to K_{24} in the routing table of N_4) and then forwarded to N_{24}. The figure also shows the leaf sets and routing tables of nodes N_4 and N_{28}. Note that some rows are

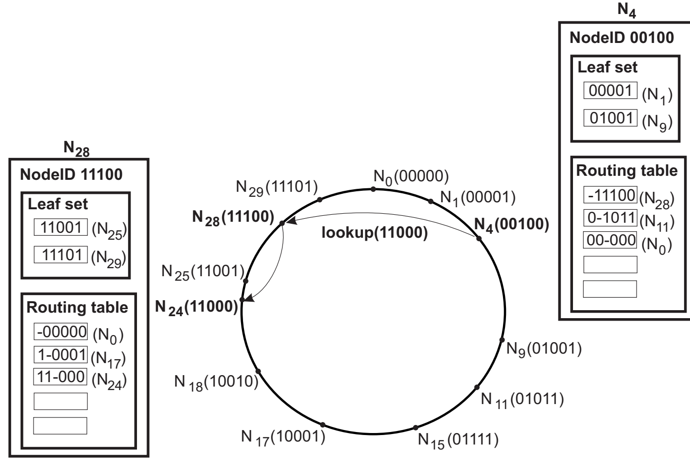


Figure 2.3: Pastry overlay structure and routing principle.

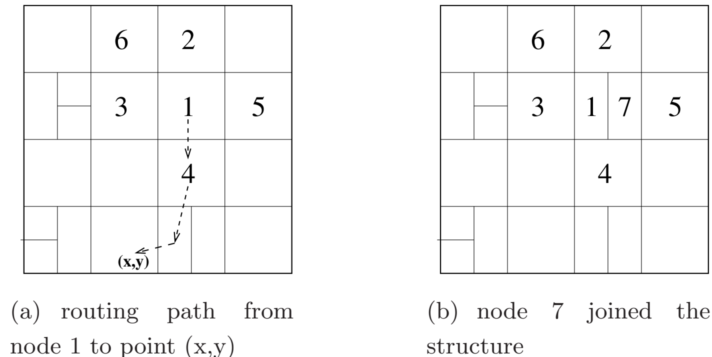
empty in the routing tables, since there are no nodes in the system to fit them. Also as an observation, in a single hop a request does not necessarily go one digit closer to the destination: for example, from N_4, which did not share any prefix with K_{24}, the request is sent to N_{28}, which shares the first 2 digits with K_{24}.

Pastry uses thus prefix routing and it has the particularity of allowing a large choice for the nodes in the routing table (especially in the first rows). We use Pastry in our research as a DHT system model to elaborate on routing tables that take into consideration the load on each node. Among all possible nodes for an entry, we choose the nodes that are less loaded, in order to balance the routing load in the system. Our algorithms are described in detail in Chapter 5.

2.1.3 CAN: Content-Addressable Network

CAN [67] is a DHT that has a d-dimensional coordinate space divided in N zones. Each zone is assigned to a node and two nodes are neighbors if their coordinates overlap over the same $d-1$ dimensions and abut along one dimension. Each node maintains thus $2d$ neighbors. Each key identifier represents a point in this space, a key being thus defined by its coordinates.

Figure 2.4 shows an example of a CAN overlay with 21 zones. Only some node identifiers are depicted in the figure. Node 3 is considered a neighbor of node 1 because their coordinates overlap over the Y axis and abut along the X axis.



(a) routing path from node 1 to point (x,y)

(b) node 7 joined the structure

Figure 2.4: Example of a CAN structure, with routing and joining samples (figure from [67]).

When routing, the requests are forwarded in a greedy manner: a node forwards a request to the neighbor whose zone coordinates are closest to the coordinates of the requested key. Routing is achieved with an average path length of $(d/4)(n^{1/d})$. Figure 2.4(a) shows an example of routing in CAN. Node 1 sends a lookup request for the key with coordinates (x, y). The arrows show the path of the request, which is sent from one node to another one, getting at each step closer to (x, y).

When joining, a newly arrived node randomly chooses a point P in this multidimensional space and routes a join request to it. The node responsible for P will split its zone in two, keep the first half for itself and assign the second half to the new node. The split is done per dimension, taking them in order. For example, in Figure 2.4(b), node 7 is joining the system by choosing a point P that lies in the zone of node 1. This latter node splits its zone in half on the X axis. (If another node would send then node 1 a join request, the zone would be split along the Y axis.) After joining, the neighbors of the node 1 are notified in order to update their neighborhood lists. This procedure only affects the neighborhood of node 1, which means $O(d)$ nodes.

When a node n_a leaves the system, its zone is assigned to one of its neighbors, say n_b. The most desirable case is when the zones of n_a and n_b can be merged into a single one, for which n_b will be in charge. If the zones cannot be merged, n_b would temporarily handle both zones. A background zone reassignment algorithm takes care to avoid the fragmentation of the multidimensional space.

As optimizations, CAN allows the tuning of the number of dimensions, but also the number of coordinate spaces. A higher number of dimensions increases the number of

neighbors per node, which implies higher maintenance costs, but it has the advantage of increasing the routing fault-tolerance in the system. This means that during routing, any request is more tolerant to (link or node) failures, since with more dimensions, a node has more candidates as next hop in the routing path: even if one candidate cannot be contacted to send the request to, another one can be used instead. With multiple coordinate spaces, called realities, each node is assigned to a different zone in each coordinate space. A key is thus assigned to a number of nodes equal to the number of coordinate spaces, which is clearly increasing the chances of having at least one live node responsible for the key. Besides this, having multiple realities also increases the routing fault-tolerance: in the case of a failure on one reality, a message can still be routed through the other remaining realities.

With either multiple dimensions, multiple coordinate spaces or both, CAN has the advantage of offering multiple paths between any two points. This *asset* is key to providing not only routing fault-tolerance in the system as we have seen before, but also support for proximity routing or load balancing, as we will discuss later in Chapter 4.

2.2 Gossiping as a Means to Self-Organizing

While most DHTs use on-demand construction and repair mechanisms, some peer-to-peer systems use protocols that are entirely based on self-organization in order to produce emergent behaviour. These protocols are based on the principle that local, knowledge-limited interactions between peers lead to a global organization of the system.

Self-organization is a process of improving the organization of a system without any external influence. Both the involved resources and the constraints related to the organization are internal to the system, and they characterize the evolution in time of the system.

In this context, we give an overview of gossip-based protocols, also known as *epidemic* protocols, a class of fully decentralized protocols that are widely used for implementing self-organizing behaviors in dynamic networks. This represents the background for our network-friendly information dissemination mechanism described in Chapter 6.

2.2.1 Gossip-Based Protocols

Gossip-based protocols were first introduced in the context of database replica synchronization [16] and they have since received considerable attention, mostly due to their ability to support large scale information systems with a simple, yet efficient and robust approach. They rely on periodic, pairwise exchanges of small-size state information between peers. Their scalability stems from the balance of communication amongst peers, and from the fact that each peer only needs to know a small part of the network, which is usually called its *view* or cache. The view thus contains a list of neighbors (limited knowledge) and it is dynamic (this set can be updated). Periodically, each peer chooses one other peer in its view to perform an information exchange. The nature of this information, as well as the result of the exchange (i.e., what information is exchanged and what is eventually kept on each side), defines a gossip-based protocol. The following algorithm presents the abstract operation of the protocol at each peer:

Active thread: periodically send a gossip request (n_a)

$n_b \leftarrow$ `selectPartner()`
$\text{buf}_{\text{send}} \leftarrow$ `selectToSend()`
send buf_{send} to n_b
receive buf_{recv} from n_b
local_state $\leftarrow$ `selectToKeep()`

Passive thread: reply to incoming gossip requests (n_b)

receive buf_{recv} from n_a
$\text{buf}_{\text{send}} \leftarrow$ `selectToSend()`
send buf_{send} to n_a
local_state $\leftarrow$ `selectToKeep()`

Each node n_a runs both an *active* and a *passive* thread. The active thread periodically selects from the local view a partner n_b to gossip with by using the `selectPartner()` operation. The data sent by n_a to n_b is determined by the means of the `selectToSend()` operation. The passive thread on node n_b receives this information, merging it into its own state (by using the `selectToKeep()` operation), and optionally sends back some data to n_a, which may in turn update its state. The size of the state is usually of bounded size. When the objective of the protocol is to create a topology amongst peers, the state is usually composed of peers (e.g., [36, 87]).

`selectPartner()` : selects a partner to gossip with from the *view* of node n_a. This view can be static, constructed by the protocol itself, or proposed by another protocol. Using the latter case, one can easily stack gossip-based protocols: the

selection of a peer for one protocol is made using the view constructed by another (e.g., [58]).

`selectToSend()` : selects the data elements to send to n_b. This data is part of the state of node n_a and can be constructed by the protocol, proposed by an application, or consist of (a subset of) the *view* of n_a, such as a set of peer identifiers.

`selectToKeep()` : updates the state of node n_b using the data received from its partner, n_a. The size of the state can be bounded or unbounded. When bounded, only a limited number of data elements can be kept in the local state.

Gossip Usage. Gossip-based protocols can be used for a variety of tasks [49]. The periodic exchange of information makes the system converge towards a state with desirable properties, which we classify as being related to the overlay itself, some management operations or applications using the emerged state (some examples are displayed in Table 2.1).

Context	Emerged behavior	State (what is exchanged)
Overlay	**overlay creation and maintenance** (creation of specific overlays and maintenance of their overlay links, e.g., [36, 87])	(subset of) view
	system reliability, such as failure detection (e.g., [86])	heartbeat counter
Management	group management and slicing (e.g., [38])	specific values, according to peer attributes
	computing the size (number of peers) of the system or some aggregates of distributed values (e.g., [45, 61])	data, based on computations, and corresponding weights
Application-related	**information dissemination**	epidemic message
	semantic routing (e.g., RayNet [69])	subset of view

Table 2.1: Gossip usage.

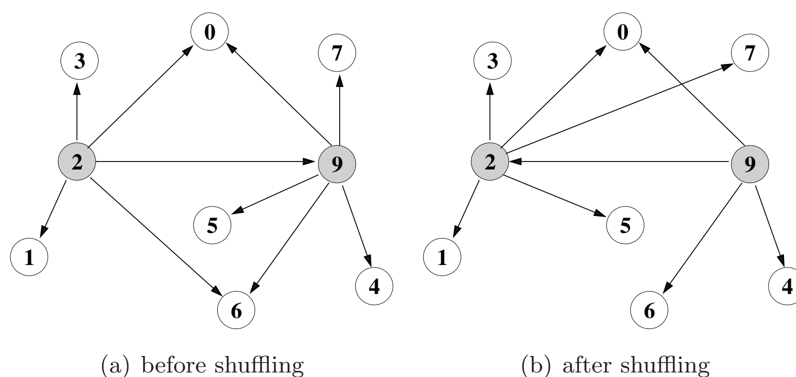
We only discuss in detail two aspects, which are relevant to our load balancing solution from Chapter 6: (1) the overlay creation and maintenance and (2) the information dissemination (shown in bold in the above classification). Nevertheless, we also give short descriptions of other relevant current applications of gossiping.

2.2.2 Overlay Creation and Maintenance

Gossip-based protocols can be used to create overlays whose structure (e.g., a random graph or a distributed data structure) emerges globally from local, knowledge-limited interactions. The overlay is not only created this way, but it is also maintained by the continuous communication between the nodes that results in dropping links towards nodes that are no longer responsive. The overlay links will eventually point towards nodes that are alive, which ensures a good connectivity between nodes.

Creating random overlays. We first discuss *peer sampling* [37, 40] protocols. Their objective is to create, for each peer, a view composed of peers samples drawn *randomly* from all peers in the network. The resulting graph, when considering bounded view sizes, is close to an Erdös-Rényi random graph [18].

CYCLON [87] is an example of a peer sampling protocol. It operates on a view of c peers. Each node n_a periodically exchanges a subset of its view, plus its own identity, with the peer n_b from its view that n_a contacted the least recently (particularity of the `selectPartner()` operation). The result of the exchange is that links from n_a to this subset are *shuffled* with the links received from n_b, in such a way that the in-degree is kept balanced. CYCLON peers are thus present in expectation in c views.



(a) before shuffling (b) after shuffling

Figure 2.5: Example of shuffling with CYCLON.

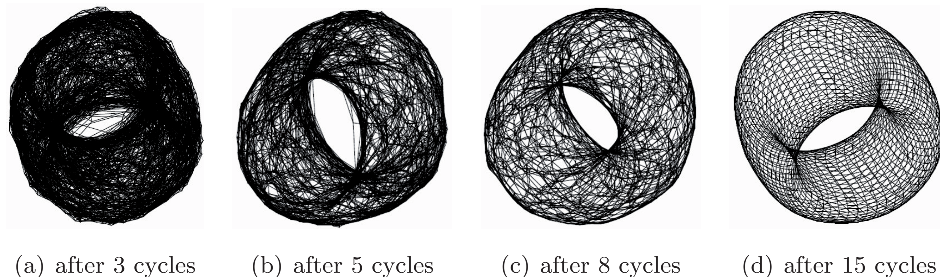
Figure 2.5 shows an example of shuffling (from [87]). Node 2 sends the subset {2,0,6} to its partner, node 9, which replies with the subset {0,5,7}. Both nodes, 2 and 9, update their views: node 2 drops the links to nodes 6 and 9, adding new links to 5 and 7, while node 9 drops the links to nodes 5 and 7, adding a new link to 2.

For both nodes, 0 was also dropped, but added again. Thus, for the in-degree of the peers that were exchanged, it remains the same for 0,5, and 7; only for 6 it decreased, since node 9 already had it in its view. Among these changes, the link between the two partners always reverses direction.

CYCLON also supports efficient membership management (addition of new nodes and removal of failed nodes). A newly inserted node issues c random walks and for each of them it initiates a shuffle of length 1 with the node where the random walk ends. This latter node takes out a node from its cache and sends it to the new node, then places the new node at the newly available entry. Departures are detected when a node fails to initiate a shuffle with one of its nodes from the cache. In this case, the non-responsive partner is removed from the cache of the initiator.

Emerging structure. The second and more adaptive kind of view management protocols construct overlays whose *structure* emerges in a totally decentralized fashion (e.g., distributed hashtables [60], semantic overlays [90]).

T-MAN [36] and VICINITY [90] are two generic protocols for expressing emerging structures. Both operate on the same principle. Each node constructs a view, usually of fixed size t, that satisfies some constraints expressed as functions over the neighbors characteristics. The goal is to make views evolve towards a set of peers whose "sum of desirability" is the highest possible, as defined by a *proximity function* (ranking function). Views are initially filled using the underlying peer sampling service. The main gossip operations are defined as follows. In VICINITY, `selectPartner()` selects as partner node n_p the least recently accessed peer from the view and `selectToSend()` picks a subset of the view by choosing peers according to three variations: random, biased (the peers from the VICINITY view with the lowest proximity scores w.r.t. n_p) and aggressively biased (the peers from the VICINITY and CYCLON views with the lowest proximity scores w.r.t. n_p). In T-MAN, `selectPartner()` selects the closest peer according to the ranking function and `selectToSend()` groups the node itself, the view and a set of random peers from the system (i.e., a random view provided also by a peer sampling service) in a single buffer to be sent. `selectToKeep()` simply merges the received elements with its own view, sorts the nodes according to the proximity/ranking function, then keeps the first t elements.



(a) after 3 cycles (b) after 5 cycles (c) after 8 cycles (d) after 15 cycles

Figure 2.6: Visualisation of a torus structure constructed with T-Man (figure from [36]).

An example of constructing a torus structure with T-Man from a completely random topology is presented in Figure 2.6. The target topology is already visible after few cycles and 15 cycles are sufficient for convergence.

2.2.3 Information Dissemination

The most common use of gossip is arguably for information dissemination [7, 19, 44]. The information spreads much like an epidemy in a human population. We distinguish two types of pairwise contamination between an "infected" and a "non-infected" peer: by *push* and by *pull*. Both models rely on peer sampling.

In a *push* model, a peer that receives a message for the first time sends it at the next round to f other peers (f is called the fanout). The message is tagged by a *hops-to-live (htl)* value, which is decreased at each peer encountered. The message is no further propagated if *htl* reaches 0.

In a *pull* model, a periodic request is sent by an active thread on peer A and handled by a passive thread on peer B. Peer A sends its set of message identifiers to peer B, which sends back the messages that A is missing. Note that pull requests can be piggybacked on top of existing gossip-based messages such as the ones used for membership management.

These models can be used as singletons (push-only or pull-only), or combined. A push-only dissemination protocol reaches all peers in the network w.h.p. in $O(\log N)$ rounds if the fanout $f = O(\log N)$ [48]. In a combined model, the peers that have not been infected by a push operation can later obtain the message by pull requests. As such, a dissemination protocol combining synchronous push and pull operations exhibits a two-phase scenario. The dissemination in the first phase is mostly due to

pushes, where the number of infected peers follows an exponential growth, while the second phase relies essentially on pulls, where, as the probability to reach non-infected peers by a push operation decreases, the number of non-infected peers decreases in a quadratic manner [7].

In Chapter 6 we present our information dissemination mechanism based on gossip protocols that not only assures dissemination completeness, but also limits the load imposed on the elements (routers and links) of the underlay network upon which it operates.

2.2.4 Other Applications of Gossiping

News service. A simple applicability of gossip-based protocols is news dissemination. The Newscast Protocol [88] is based on an epidemic-style protocol, which manages membership and information dissemination. Its functionality is very basic: the cache on each peer contains news and the IP address of their initiator, and periodically the most recent news are sent to a random peer from the cache. Going a bit further, the service described in [24] disseminates news items only to users that are interested in them, combining the functionalities of peer sampling and dissemination.

Building routing tables. Another example of gossip-based protocol used for building overlays is the epidemic protocol for managing routing tables in [89]. The idea is to create and manage routing tables in a system where the nodes and key identifiers represent sequences of digits and the routing is based on reaching the destination node gradually, by matching one more digit to the destination key (as Pastry does). The nodes exchange subsets of views using the Newscast Protocol. An instance of the Newscast Protocol (called agent) is used at each level in the routing table, where the local node and the set of neighbors (c neighbors) at level i share the first i digits. With this structure, the gossip communication is done per level, between the agents at the same level i (that share the same prefix). The agents exchange the set of nodes at that level, and the partner is chosen as one of them. It is clear that the higher levels (that have to contain nodes that share a larger common prefix with the local node), are harder to fill. As an enhancement, an agent i sends the received set to the adjacent agent on the same node that expects neighbors with a larger common prefix (level $i+1$), just in case the received nodes would better fit at this latter level.

Group membership management. Gossip-based protocols can also be used to maintain group membership. In the SCAMP protocol (Scalable Membership Protocol) [22], group membership is established by gossiping subscription requests based on the push model. A joining node n_j sends a subscription request to an arbitrary member, which forwards the request to all members in its view and, for fault tolerance, additionally to randomly chosen c members of the same view. Then, a node that receives the request will either forward it to a random neighbor or it will add n_j to its view (if it does not have it already), with a probability that is inversely proportional to the size of its view. This choice has the advantage of balancing the view size of all nodes, without global knowledge of the group size. In order to silently get rid of old subscriptions, each subscription has a lease time; when it expires, a node holding the subscription deletes it. Node n_j will also keep the list of nodes that added it in their views.

In [38], peers are grouped according to some of their attributes (as for example available bandwidth) using an ordered-slicing algorithm based on gossiping. In order to achieve that, all peers generate random numbers, which are then swapped in the periodic gossip exchange if the peer order differs when comparing their attributes and when comparing their random numbers.

Computing aggregate functions and the size of the system. In distributed systems, computation of aggregate functions is a fundamental process. Data resides on different nodes in the system and their aggregation needs to be known either by the initiator of the aggregation or by all peers. Computing the sum, mean or variance of some values hold by the nodes of the system can be easily accomplished with epidemic protocols. Several such solutions are proposed in [39, 45, 46].

An information that is not readily available to a distributed system, but that might influence its performace or functionality, is the knowledge of the number of its participants. Using the protocols that compute the mean while having a value of 1 at the initiator node and a value of 0 at the other nodes and expressing the mean also as $1/N$, the computation of the size N of the system (i.e., its number of nodes) is straightforward.

Alternatively, a simple addition [52] to a peer sampling protocol such as Cyclon allows to make the estimation of the size of the system. Each node gets a random iden-

tifier and it hashes it into a key space. Next, collecting the closest hash values around a node's identifier and considering their density, one can obtain a sound estimation of the system size with a very limited overhead.

Content distribution. Epidemic schemes are also used in collaborative data delivery, where multiple chunks of data that reside at different nodes need to be delivered to all nodes in the system. In order to assure complete delivery at all nodes, an epidemic protocol is proposed in [20], which is based on a coloring mechanism. In this protocol, each node has a unique color that indicates the chunk for which it has primary forwarding responsibility. Nodes exchange chunks and the partners are chosen based on the nodes colors, ages, the chunks they hold and randomization.

Clock synchronization. The Gossiping Time Protocol [35] (GTP) gossips time information to synchronize clocks to a system time source. The condition for a local clock to be updated with information received from a peer is based on hop count and dispersion metrics. GTP uses the peer-sampling service to select the peer to gossip with, while the frequency of gossiping is tunable.

Predict the best search technique for a query. Gossip-based protocols are employed in hybrid peer-to-peer networks in order to detect the best search technique for a query [96], between flooding or a lookup in a DHT. Nodes exchange synopsis information of the documents in the system, and the detection of the search technique is based on the resulting global statistics.

2.3 Summary and Discussion

In this chapter, we have introduced peer-to-peer systems and we have detailed some of the most known DHTs: Chord, Pastry and CAN. We have discussed their characteristics and their connection to our work. Additionally, we have approached self-organization in peer-to-peer systems. We have given examples on the outcomes of gossiping, with a bias on information dissemination.

As we have seen, DHTs differ mostly in their structure, rules of associating objects to peers and routing strategies. Their common goal is information lookup, for which they focus mainly on providing fault tolerance. When routing, they use a different

neighbor in case of failures (e.g., lower entry for Chord and Pastry and a node from a different reality for CAN). Moreover, they support replication to ensure that objects lay on at least one live node. Load balancing is also considered, however mostly regarding the namespace through the hash function that they employ to assigns identifiers. But for routing load balancing, they do not provide specific solutions. Chord uses greedy routing, which is not appropriate for balancing the load, as we will see in the following chapter. However, we have noticed that Pastry may offer the support for routing load balancing through the flexible choice of the neighbors and CAN through its multiple realities. For this reason, we propose a routing load balancing solution based on a Pastry DHT that we will be presenting in Chapter 5. Moreover, we will be designing our own DHT in Chapter 4 for better support for routing load balancing.

For self-organization, gossiping has many applications: content distribution, clock synchronization, DHT creation, etc. Gossiping is based on periodic information exchange between any two peers. The existing protocols differ in the nature of the information being exchanged, the choice of the partner peer to gossip with, the choice of the information to be sent and the aggregation method between the local and the received information. Two operations are defined for information dissemination: push and pull, to send and request an information, respectively. The parameters of these operations (the views that they use and their fanout) determines not only the speed of the dissemination, but also its impact on the underlying network. This latter issue will be tackled in Chapter 6.

This chapter serves as background for our load balancing survey and for our load management solutions for information lookup (DHTs) and for gossip-based information dissemination.

Chapter 3

Load Balancing in Peer-to-Peer Systems

"A balanced diet is a cookie in each hand."

Anonymous

3.1 Introduction

In this chapter we survey current load balancing solutions in peer-to-peer systems, to serve as a background when presenting our solutions in the following chapters.

We settle this survey in the area of wired networks, where each peer can communicate with any other peer from the system (using their IPs). Thus, they allow for a wider choice of load balancing solutions when comparing to mobile systems, which are usually limited by a range of reachability.

Peer-to-peer systems are set up as overlays which rely on an existing underlying network. Two peers are connected through a logical link at the overlay level, which is actually a routing path at the underlying network level. Since we aim at load balancing issues related to the peer-to-peer system itself (i.e., its infrastructure and functionality), we do not interfere with the underlying network management and simply consider the underlying network as an IP best-effort network.

From a user's perspective, peer-to-peer systems are expected to allow any peer to join and, once connected, to offer a good response time for any user interaction (e.g., for a query). The overlay management thus has to deal internally with finding a path

between any two peers, resource load balancing, tolerance to peer and link failures, scalability of the system to allow it to grow or shrink, connectivity, security aspects etc. When providing a better solution for one of these issues, usually they are considered in a global context, having the impact on the other matters also considered; in case of a negative impact, this should be rather small and justified. For example, providing a solution for a shorter path does not necessarily mean that it is also fault tolerant or that the load on the resources of the system is still balanced.

Being decentralized, this kind of systems are easily prone to a lack of balance in the utilization of system resources. An unbalanced mapping of the objects over the nodes, an unequal in-degree of the nodes or a bad choice of the path to follow for requests are just some examples of situations that can easily lead to some resources being used more than others. In peer-to-peer systems there is no central entity to equally distribute the load on peers and links. As a consequence, distributed load balancing algorithms are applied in the functionality of the system, not only to provide fairness for resource utilization, but also to achieve an optimal global utilization of the resources.

Load-related terms. There are specific terms that are related to load balancing and which are often used in research papers. The *load* can either be related to objects, peers or links. An object that is requested very often is said to be *popular*. The object popularity, object size and the number of objects on a node form the *object load*. Each node has a fixed *capacity* which might represent available disk (storage) space, processor time or bandwidth [26]. The load generated by the request processing of the lookup queries that end at a node (i.e., the receipt of a lookup query and the preparation for delivery to the inquiring node) is referred to as *request load*, while the load generated by the forwarding process of the lookup queries that are sent further away at a node is called *routing load* [76]. The traffic in the overlay and underlay is called the *traffic load*.

In this survey, as support for our contributions, we cover related work on load balancing in Distributed Hash Tables (DHTs) and on network-awareness as mean to balance or reduce the load in DHTs or less restrictive overlays. In Section 3.2 we give an introduction on load balancing causes and solutions. We then present the solutions per categories, in the following sections: Section 3.3, related to object placement, Section 3.4, which focuses on the traffic generated by the requests and routed in the

system, and Section 3.5, where the underlay network is taken into consideration. Last, we make a summary and a short discussion on the presented solutions.

3.2 Preliminaries on Load Balancing

In this section we present an overview on DHT overlays and the load balancing problems that may appear during the lookup procedure. Nevertheless, some of these problems are also common in other types of systems: object load, some routing issues, the generated load in the underlay. As a prerequisite for efficient load balancing, we start with a short discussion on the DHT overlay structure and maintenance. We then present the main causes of imbalance and we classify their solutions.

3.2.1 DHT Overlay Guidelines

A "good" overlay is an overlay that is well balanced under a uniform flow of requests. The namespace, routing tables and routing strategies are thus defined accordingly. In the following paragraphs we present the characteristics of a good overlay, giving some insights on how to populate its structure, how to create and maintain the routing tables and how to take into consideration the underlying topology.

Creation of the structure. Each node and object has an ID in an identifier space. These IDs and the links between the nodes form the overlay structure. Obtaining a uniform distance between the nodes on the identifier space and a uniform assignment of the keys to the nodes is a very important issue in order for all nodes to be considered equal. This is widely known as *namespace balancing*. The identifiers may be obtained from a hash function: for a node usually its IP, while for a key usually its name, is hashed into a sequence of bits of a specific length. Using a good hash function that balances the namespace is called *consistent hashing*. Alternatively, there can be other ways to assign identifiers to nodes, such as random positions [47], Hilbert numbers [83], grouping by IP-addresses [23], or using some node from the overlay as identifier generator [79]. Section 3.3.1 discusses namespace balancing.

Creation and maintenance of the routing tables. Each node maintains a list of neighbors in its routing table (i.e., outgoing links), and optionally the list of nodes

that have it as a neighbor (i.e., incoming links). The first observation related to load balancing here is that a node that has a small number of incoming links will receive on average less requests, as noted in [95]. In order to allow a fair share of the traffic for each node, the routing tables are created in such a way that the number of links per node is balanced, i.e., most of the nodes have on average the same number of incoming links and a node has roughly the same number of incoming links as outgoing links. Moreover, it is well known that having *redundant paths*, as multiple neighbor choices for a single routing strategy, or different routing strategies, one could balance better the routing load [79] by sending the requests through different paths towards destination. Thus, for both fault tolerance and load balancing, the routing strategies should allow the possibility to choose between several entries of the routing tables when sending a request, and guarantee that the request reaches its destination regardless the chosen entry, at least when no failures occur.

The typical maintenance mechanism of the routing tables is by periodical update: when neighbors are detected to be dead, they are replaced with new live nodes. The new nodes have to conform to the same routing table rules, such as restrictions on the identifiers and possibly on their position at underlying level according to the predefined metric.

Underlying topology. The *underlying topology* knowledge has also to be considered from the design time of the overlay, since this is either reflected in the identifier assignment or in the creation rules of the routing tables. There are overlays that take into consideration some distance metric on the IP level (e.g., the delay time, the number of hops): in Pastry [71], the neighbors of a node are chosen among the nodes that are reached in the lowest delays or shortest path, while in a version of CAN [68], nodes are grouped in bins based on on-line measurement techniques. In TOPLUS [23] the nodes are grouped based on their IP addresses.

3.2.2 The problem: causes of imbalance

The causes of load imbalance are very various: an imbalance may occur in all resources of the system. Table 3.1 shows the main causes of load imbalance, detailed per context in the following paragraphs.

Context	Cause Description	Load
Overlay namespace	unequal namespace or number of keys assigned to nodes	number of keys per node
Requests	popular keys	request load
Routing	links or sequence of links used very often for routing	routing load
Underlying topology	routing without knowledge of the underlying topology	underlying traffic load

Table 3.1: Causes of load imbalance.

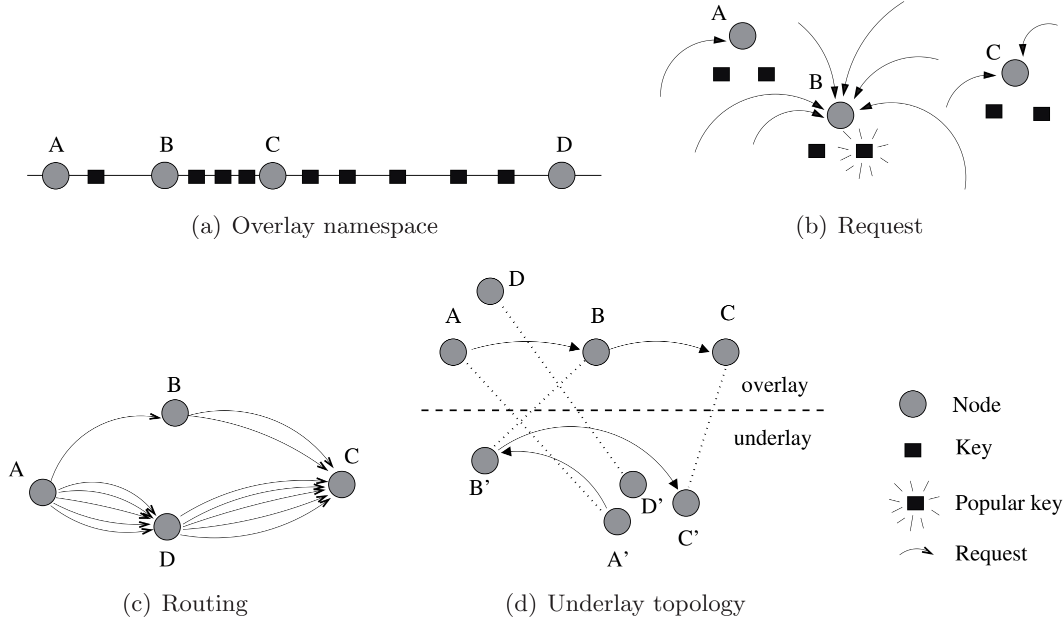
Overlay namespace. All nodes and keys have identifiers in an identifier space (also called name-space or address-space). A wrong placement of the nodes and keys on the identifier space can cause an unequal address space assigned to each node, which can easily generate an unequal number of keys that nodes have to be in charge of. Here, the load is the number of keys per node.

Requests. Some keys are popular, being requested much more often than others for some period of time. Consequently, the nodes that own such popular keys, and that have to deliver them to the nodes requesting them, will be more loaded. The load is the number of requests for the keys that a node owns (i.e., request load).

Routing. The routing strategy decides which neighbor (i.e., node from the routing table) is chosen as the following step for a request. When the routing strategy sends most of the requests through the same neighbor, that neighbor and the corresponding link are prone to become more loaded than other nodes/links. Moreover, if this happens at consecutive nodes on the request path, the path will become burden by all these requests. The load represents the number of requests that travel on a certain link or path (i.e., routing load).

Underlying topology. When the overlay has no notion about the underlying topology, the requests might follow too long paths in the underlying network, generating additional traffic.

Some scenarios that illustrate these imbalance causes are shown in Figure 3.1. Scenario (a) is related to the overlay namespace, that we have represented with a line. A, B and C are nodes, and the little squares represent the keys. We consider that each node is responsible for the keys at its right, until its successor. In this figure, we



(a) Overlay namespace (b) Request (c) Routing (d) Underlay topology

Figure 3.1: Scenarios with imbalanced load.

observe two problems. (1) Even though the size of the namespace of nodes A and B is the same, their number of keys is not balanced, node B having more keys. (2) The size of the namespace of node C is much bigger than the namespace of node A or B, and thus, even if the assignment of the keys would be balanced on the identifier space, node C has much more keys than them.

Scenario (b) deals with request load. In this example, each node is responsible for two keys, so the number of keys is balanced on the three nodes. In particular, node B owns a popular key, so B will receive much more requests (the curved lines) than the other two nodes, making B far more overloaded than them.

Scenario (c) is about routing. Here, the overlay deals with a large traffic between nodes A and C. This traffic can pass either through node B or node D. The problem here is that most of the requests prefer to pass by D, and thus the path A $\rightarrow$ D $\rightarrow$ C is far more loaded than the path A $\rightarrow$ B $\rightarrow$ C.

In scenario (d), the overlay has no knowledge about the underlying topology. Four nodes are represented at both overlay and underlay levels. The correspondence between the same node at overlay and underlay level is shown with dotted lines. We use curve lines to depict the path from A passing through B in order to arrive at C. When we

look at the path at the underlay level, we see that the path is quite long and especially inefficient, since a simple short jump from A' to C', or the correspondent of B at the underlay level to be D', would have less loaded the network.

The load on a node may vary greatly over time since it can be expected that the system experiences continuous insertions and deletions of objects, skewed object arrival patters and churn [26]. Moreover, the flow of requests may change in time. There might be *uniform* requests, when any node is requesting any key from the system, or *Zipf-like* requests, when some keys are much more popular than others. The former case is the most desired one, since load balancing might come by default under a well constructed overlay as described before. The latter case, however, is more complex. This is also the reason why the load balancing solutions have to be adaptive and should be applied all along the life time of a system.

3.2.3 Classification and applicability of load balancing solutions

There are two general types of load balancing solutions: overlay-specific (applicable only on specific overlays) and the overlay-independent (applicable on any overlay and usually initially designed for self-organizing behaviors). The former might be more efficient, however they are limited, while the latter can also be applied to any overlay as complement to an existing overlay-specific or -independent solution in order to improve its results. Examples of overlay-independent load balancing solutions include caching and replication, and the usage super-nodes. The super-nodes are usually nodes of higher capacity that have been for a long time in the system, that are delegated for example to alleviate loaded nodes by diverting the traffic instead towards less loaded nodes.

For most of the load balancing solutions, if a highly-loaded node is alleviated, by default also some of its incoming or outgoing links will be alleviated. There are no specific solutions for exclusively alleviating the nodes or exclusively alleviating the links, and neither should be a reason for doing this.

The load balancing approaches can be mainly applied for the namespace, the request rate for popular objects and the routing traffic at overlay and underlay levels. More detailed, they are grouped into three categories:

Category	Solutions	Nodes own too many objects	Nodes serve too many requests	Nodes forward too much traffic	Too much or unbalanced traffic in the underlay
Object Placement	Namespace balancing	X	□	□	-
	Virtual Servers	X	X	-	□
	Multi Hashes	X	-	□	□
	Caching and Replication	-	X	□	□
Traffic Routing	Link Reorganization	-	-	X	□
	Path Redundancy	-	-	X	□
Underlay	Proximity Awareness	-	-	-	X

Table 3.2: Load balancing solutions.

- Object Placement, which deals with the positioning of the (nodes and) keys in the identifier space, the mapping of keys to nodes, the physical placement of the objects and the request rate especially for popular objects (see Section 3.3);
- Traffic Routing, which is directly related to routing strategies and routing tables (see Section 3.4);
- Underlay, which deals with the network traffic at the underlying topology level (see Section 3.5).

Table 3.2 shows an overview of the existing load balancing solutions presented in this chapter. An "X" means that the solution is applicable for the specified problem, a "□" means that it not meant to, but it is applicable to some extent, and a "-" means that it does not solve that problem. These solutions are discussed in the following sections.

3.3 Object Placement

In this section, we present the load balancing solutions that achieve the following goals:

- assign an equal address space to each node: *namespace balancing*
- assign objects of equal size to each node: object load balancing by *virtual servers* and *multiple hash functions*;
- balance the request rate for a certain object, usually popular, over several nodes: *caching and replication.*

In order to get an ID on the identifier space, the mostly adopted method for the keys is to use a hash function. The assignment of the keys to the nodes is often linked to their position on the identifier space: a key is either assigned to the following node on the identifier space (i.e., the first node with an ID larger than the key), to the predecessor or to the closest node. Whether the namespace is not balanced or the size of the objects is not uniform for all nodes, the load can be transferred from one node to another using virtual servers. In order to avoid load transfer, several hash functions can be used to propose different locations for an object, and only one is chosen either based on the load of the nodes at those locations or using some other criteria. When an object is very popular, thus its owner receives too many requests for it, the object can be replicated on multiple places (i.e., nodes) in the overlay using caching and replication. These solutions are further detailed in the following subsections.

3.3.1 Namespace balancing

As stated in the previous section, there are several mechanisms to assign identifiers to nodes and objects. Consistent hashing is a form of namespace balancing that has been introduced by *Karger et al.* in [42]. It uses a good hash function that assigns identifiers for each new node and key in an uniform manner on the identifier space. The keys are then assigned to the nodes. With consistent hashing, when a node leaves the system, only the keys that he owns will have to be assigned to other nodes; no other reassignments are necessary.

Most of the current research is based on a uniform load assumption [27]: the load of each node is proportional to the size of the address space it owns. However, as

part of the assignment process, when a node n becomes responsible for a key k, there are two things that can be transferred to n: either the whole object with ID k or a redirection pointer towards the IP address of the owner of the object. Hence, it is clear that consistent hashing may generate a load balancing problem when redirection pointers are not used and the objects have different sizes [82]. Some nodes might end up only with objects that are big in size, while others only with small objects.

The other mechanisms that assign identifiers to nodes and keys and that are meant for namespace balancing are mainly based on a pre-check of interval length. In [47], a newly joining node chooses r random points in the identifier space, checks the length of the intervals of the neighboring nodes for each point and then selects its position at the half of the longest interval. For more uniform balancing but at the price of generated churn, in [6] each node is responsible for an interval and periodically nodes that have short intervals are forced to leave, in order to join somewhere else, in a larger interval. A node knows whether its interval is short, long or middle, based on the length of the entire address space and an estimation of the number of nodes in the system. For the latter, each node makes a marker in the identifier space (contacts the node responsible for a random identifier in the system); then, each node counts the number of markers on its interval and communicates with its successor in order to get the estimation. This solution reduces the *smoothness* (i.e., ratio of the length of the longest interval to the length of the shortest interval) at the cost of the traffic generated by the estimation of the number of nodes in the system (but which can be used also for other purposes) and the generated churn induced by the forced leaves and joins.

3.3.2 Virtual Servers

The virtual servers were first introduced by *Dabek et al.* in [14]. Instead of only one position, each node has multiple instances on the identifier space, each one acting as a node. Each such instance is called a *virtual server*. In order to balance the load (i.e., the total size of all objects that a node owns), nodes exchange virtual servers, minimizing the amount of moved load from one node to another. This kind of solution gives flexible choice: a node may choose to keep active in the same time either several virtual servers or only one.

Rao et al. [66] have defined three schemes, where the nodes contact each other in order to exchange virtual servers. For balancing the load, each node that is *heavy*

(i.e., highly loaded) assigns one of its virtual servers to a node that is *light* (i.e., less loaded). The three schemes are presented in increasing order of their efficiency (but also complexity). The first scheme is *one-to-one*: a light node contacts random nodes until it finds a node that is heavy. It is for the light node to do the probing, in order not to charge the heavy nodes also with the probing mechanism. The second scheme is *one-to-many*: a heavy node will transfer one of its virtual servers to the most suitable node among several light nodes. The process is based on directories whose identifiers are based on a special hash function and each node is contacting a random directory where it reports its load. When the node that owns a directory is contacted by a heavy node, it replies with the most suitable light node and the transfer may begin. The third scheme is *many-to-many*: again, each node contacts a random directory and reports its load, however now it is for the node that owns the directory to check for suitable transfers between heavy nodes and light nodes. These checks are done periodically. The most effective scheme in terms of exchanged loads is clearly *many-to-many*, since the directory chooses from both sets (the heavy nodes set and the light nodes set) the best transfers to be done, at the price of relying on centralized directories.

Later, *Godfrey et al.* [26] have complemented the solution of periodic reassignments of virtual servers, by using directories with an emergency balancing request. As before, each node chooses a random directory where it reports its load. After being involved in a set of transfers, it will report its load to another random directory. The emergency request is issued by a heavy node, when its load has run on top of its threshold, and it is targeted on overcoming this kind of critical situations. Under no critical situation however, initiating transfers periodically remains the best solution, due to the high choice of possible transfers.

The idea of a node having several virtual servers is refined by *Godfrey et al.* in [27], by allowing only a specific range on the identifier space for the positions of the virtual servers of each node. The overlay is called Y_0 and it is based on Chord. A node v chooses $\Theta(c_v \alpha)$ virtual servers, where c_v is its capacity, $\alpha = \Theta(\log n)$ and n the number of nodes in the system. Their identifiers are chosen randomly from $\Theta(\log n)$ consecutive intervals of size $\Theta(1/n)$, starting from a fixed random point. The routing tables are maintained as if each node would have owned all the identifier space between its first and last virtual server, while the successor list contains nodes that succeed each of its virtual server. As a deduction, the advantage over choosing randomly over the identifier

space or with hash functions is that a physical node would have only one routing table instead of one per virtual servers (as previous solutions). However, for routing, besides the routing tables, the lookup scheme also needs to use the successor links for the request to arrive at its owner. Whenever the estimation of the number of nodes or its capacity changes, a node updates the number and the positions of its virtual servers. Each load reassignment is done by a leave followed by a join under a new identifier for its virtual servers, starting from roughly the same fixed point, which reduces the cost of object movement.

Karger et al. [43] use also the idea of a node having multiple possible identifiers, as with virtual servers. They call them virtual *nodes*, since a node activates only one such possible position, i.e., one node has only one virtual node active at a time. The choice of the virtual node to be active is meant for namespace balancing. This scheme improves consistent hashing by making each node responsible for a $O(1/n)$ fraction of the address space. However, by changing the virtual server to be active, it means also that the routing table has to be changed. Complementary, they also suggest a solution for transferring load, where the nodes ”move” in the identifier space. Each node i contacts a random node j and they perform a load balancing operation if node j is lighter: node j gives up its identifier and will take a new identifier close to node i in order to take some of its load.

Ledlie et al. [53] apply for each node the k-Choices algorithm to select at most $\kappa/2$ virtual servers, until the node has reached its target workload. When joining, a node probes the overlay to see the position where it could fit best; the successor node of that identifier will share its load with the new virtual server. After joining, a node can still probe the network to find a new location; such a node is called an *active* node. Active nodes periodically choose the virtual server with the maximum mismatch between its work and capacity, since loads may have changed over time. Conversely, *passive* nodes do not do any more probing after joining, so no additional churn is induced through virtual server relocation. However, to balance the load, natural churn is required in this latter case.

Table 3.3 resumes the differences between the characteristics of virtual server (VS) solutions for load balancing through object placement. The ”-” sign means that the corresponding information is not supplied by the authors or it is not relevant. As we deduct from these solutions, the idea of virtual servers is simple, each node having one

Characteristic	***Rao et al.*** [66], ***Godfrey et al.*** [26]	***Godfrey et al.*** [27]	***Karger et al.*** [43]	***Ledlie et al.*** [53]
Name	-	Y_0	-	k-Choices
VSs per node	any	$\Theta(c_v\alpha)$	1	$\kappa/2$
Position of VSs	free to move anywhere	from a fixed position per node	free to move anywhere	free to move anywhere
Path Length	-	O(log$n$)+ $\Theta(1)$	-	Ø(log$n$)
Traffic	probing mechanism to find heavy node, or 1 message to contact directory	generates churn: node leaves and then joins for its VSs to get new IDs	generates churn: node leaves and then joins to take a new ID where appropriate	only active nodes generate churn: a VS leaves and then joins to take a new ID
Central entities	relies on directories	No	No	No
Routing tables	one per VS	one of $\Theta(\log n)$ size	one per VS	one per VS

Table 3.3: Comparison between load balancing solutions that use virtual servers.

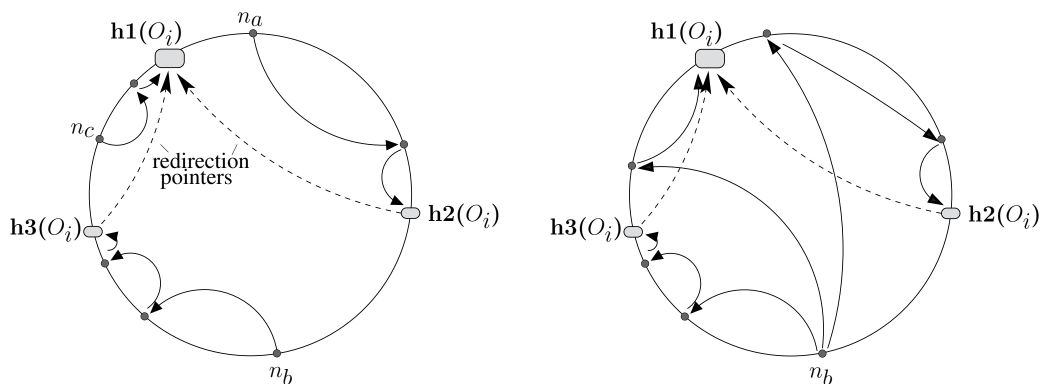
or more active virtual servers, having thus one or more IDs on the identifier space. Not only that the virtual servers of a node change over time, but also their number can change. The virtual servers might be supported to move to any position or they might be restricted to a certain interval in the identifier space in order to reduce the costs for the routing table maintenance and the object transfer as we have seen in [27]. The request path length is not an issue to any of the existing solutions, however the size of the routing tables might increase, or several routing tables might have to be considered per node. All solutions generate churn or need natural churn in order to move the load from one node to another. Some of them rely on central entities that decide when and where the load should be moved, or it is a decision taken upon the result of the communication between any two nodes. The transfers can be initiated by any node or by directories, either periodically or upon emergency request.

3.3.3 Using multiple functions to hash an object

Byers and al. [10] use a simple approach for balancing the object load by employing several hash functions and redirection pointers. An object is hashed through d hash functions, and the peer with the lowest load is chosen to accommodate this item. The authors do not state this, but for heterogeneity, one should rather choose the peer with the highest available free capacity instead. All the other $d-1$ peers hold redirection pointers towards the chosen peer. For the lookup procedure, either d parallel lookup requests are launched towards the d hash values of the object, or only one lookup request, choosing randomly the hash function. The former generates a lot of traffic, but it is obviously more fault tolerant. The latter generates only one more hop on average, comparing to the classical lookup scheme where the responsible node owns the object. The extra hop happens when the request arrives at a node with a redirection pointer, so one more hop is needed for the request to arrive at its destination. Still for the latter case, if the request fails before arriving at a node with a redirection pointer, it does not necessarily mean that the peer owning the object does not exist anymore. Again, it is not stated, but it is straight forward to see that in such scenario, to improve fault tolerance, one can sequentially choose another hash function until all d functions have been used.

The authors also present two other usages of redirection pointers: load-stealing and load-shedding. The light peers that hold redirection pointers to a heavy peer p will try to steal some of its load, or peer p might shed some load to them. This way, the lookup procedure using the predefined hash functions still holds. Alternatively, any other peer from the overlay might be used for either load-stealing or load-shedding, as either light or heavy node, but then, yet one more redirection pointer is created, which means one more hop for the request path. To improve fault tolerance, they also suggest making replicas at the $d-1$ peers (which would also remove the extra hop), or additionally at any other light peers towards which redirection pointers are then created.

Figure 3.2 shows an example of a system that uses three hash functions, where the requests are routed clockwise on a ring structure. Object O_i is assigned to the node with the lowest load out of the nodes responsible for the resulting hash values, which in this example is the node responsible for $\mathtt{h1}(O_i)$. The dashed lines represent the redirection pointers, while the filled lines are the lookup requests. In 3.2(a), nodes



(a) lookup requests from three different sources: n_a, n_b and n_c

(b) parallel lookup requests from the same source n_b

Figure 3.2: Examples of usage of multiple hash functions.

n_a, n_b and n_c issue lookup requests for object O_i, using hash functions h2, h3 and h1, respectively. The first two will have to follow as last hop redirection pointers to reach the destination. Conversely, in 3.2(b), for fault tolerance, node n_b issues three parallel requests, for each one using a different hash function. The requests that succeed will reach the destination either directly (when using h1) or through a redirection pointer (when using h2 or h3).

The idea of using multiple hash functions and redirection pointers has also been used previously in CAN [67], a structured overlay that we overview in Chapter 2. In CAN, the selection of the hash function to be used depends on the distance in the identifier space from the requesting node to the d nodes: the lookup request will be sent to the closest one.

3.3.4 Caching and Replication

Even under a uniform placement of the objects, the (fluctuating) popularity of certain objects may cause an imbalance in the request load: the nodes owning popular objects have to reply to much more lookup requests. In order to distribute the request load between several peers, multiple copies of the same object are periodically distributed to peers over the system by *caching* and *replication* mechanisms. Since the object is expected to be found faster (i.e., in a smaller number of hops), the new average lookup

path length is shorter.

With caching and replication, a peer is keeping locally a copy (a replica) of an object that is owned by another peer from the system. The main difference between them is that caching is mainly related to object request, bringing the object closer to peers that request it, while replication is mainly related to nodes that want to push copies of their objects to other nodes, updating them when necessary. A peer that has a copy of an object is called a replica peer and can act as server for that object. This means that the request load for that object is now scattered over multiple nodes.

The idea of copying objects to other locations is simple, and moreover, these two techniques are orthogonal to other load balancing solutions, which makes them suitable as complementary solutions. The solutions that we overview in this section are either based on the existing overlay structure [65, 83], they define the structure according to load balancing restrictions [34, 94], or they can be applied to any peer-to-peer system [5, 81, 91]. We further detail them in the remaining of this subsection.

Replication is usually needed when load tends to achieve its highest value [28], however there are other strategies that do not take popularity into consideration. The main challenges when doing caching and replication are:

- which objects to replicate;
- which peers should be replica peers;
- how to balance the request load between the replica peers.

Which objects to replicate. There are several choices when deciding which object should be replicated [12]. In the *uniform* replication scheme, all objects from the system are equally replicated, which also gives the advantage of any object being found faster. However, this generates a high degree of low replica utilization for the objects that are hardly requested or not requested at all. Hence, the disadvantage is that for these objects it is not worth it to use the storage resources. Much more employed is the *proportional* replication scheme, where only the most popular objects are replicated. Furthermore, an object is replicated at a number of peers that is proportional to the frequency of requests for that object. This latter scheme does not generate useless replica objects, however it is clear that the non-popular objects are much harder to find. As an in-the-middle solution, in the *square-root* replication scheme the number of

replica peers is proportional to the square-root of the request rate of the corresponding object. A study on these schemes is furthermore presented in [56].

The popularity of a node is determined usually through observed load [28, 94], but also through collected information, as in [92], where a decentralized algorithm uses random walks with a limited TTL to collect information in order to detect whether objects are poorly or well replicated.

Giving the fact that the storage capacity is limited, a caching replacement policy needs to be employed. The solution in [95] replicates objects favouring new ones over the old ones, while still taking into consideration the utilization rate of each replica object. In [5], the object to be replicated is the most popular object on the node, where its popularity is computed as a weighted moving average of its previous popularity values.

Which peers should be replica peers. It is clear that having an increased number of replica peers is a good way to offer faster access to the requested objects and some implicit fault tolerance and request load balancing. However, not to waste storage resources, only some of the peers are selected as replica peers.

For the selection of the replica peers there are mainly two replication strategies [56]: *owner replication*, where the object is replicated only at the requesting node, and *path replication*, where the object is replicated at the nodes along the path between the node that delivers it and the requesting node.

Owner replication is the easiest to employ, since it only involves two peers. An example of applicability is in [92], which treats the case of range queries (requests that contain specific constraints). When a response to a range query contains an object that does not have enough replicas comparing to the other objects in the response, that object will be replicated at the requesting node.

For path replication, in order not to replicate the object at too many peers, only some of the peers from the request path are selected. An example of a path replication strategy is shown in Figure 3.3, for a system where the lookup requests travel clockwise on a ring structure. A lookup request for an object owned by n_d goes from an initiator node n_s following the intermediary hops n_1, n_2 and n_3, until reaching n_d (the filled lines in the figure). The replica peers are chosen from these intermediary nodes: in this example, the chosen replica peers are n_2 and n_3. Node n_d will issue then replication requests (the dashed lines) in order for them to become replica peers.

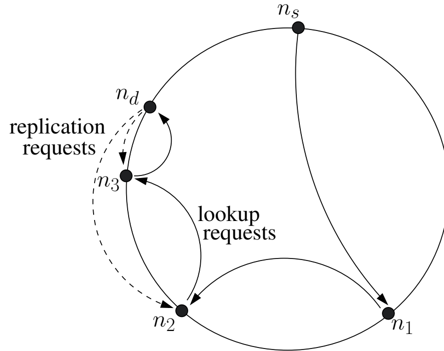


Figure 3.3: Example of path replication.

In [91], the selection of the peers along the response path to become replica peers is done according to an identifier. Each peer has a group identifier which is chosen at random from a predefined range 0..(M-1). When the object is found, its name is hashed on a value i in the same range 0..(M-1) and only the peers on the response path that belong to group i will be selected to hold a replica of that object. Moreover, for the lookup procedure, a request for an object with a hash value of i is preferred to be sent towards peers that belong to group i, in order to increase the chances of finding the object in a short time.

In Plaxton-type overlays [64], the replica peers are chosen according to their identifiers, selecting the nodes whose IDs match most closely the requested object ID. Hence, in PAST [72] or Beehive [65], a popular object is pushed for replication at decreasing levels i of prefixing with the popular object (i.e., nodes that share the same prefix of at least i digits with the object). The LCP-replication (i.e., replication in decreasing order of the length of the common prefix) has been shown to give a good lookup performace in [94] on a multiple-choice random Plaxton network, where a node prefers as neighbors the nodes with a small number of incoming links. This intent of balancing the incoming degree improves the replica distribution. Likewise, but this time for a lookup tree structure built at each node [34], objects are replicated at children nodes of the node owning the object.

Taking into consideration the flow of requests, in [5] the replica peers are chosen among the nodes that forward the most of the traffic as last hop before reaching the destination or a replica peer, which would easily adapt to any structured overlay employing

any, but only one, routing strategy.

All these solutions delegate peers as replica peers, however a peer is not always forced to be a replica peer. This is also the intuitive usual case with caching. For replication, when a peer receives a replica request, the decision to create a local object replica may be based on some own local information [95] or on its experienced traffic taking into consideration object popularity fluctuations [81]. For dropping a replica object, a peer may use the same information. When the decision is not local, the available capacity and physical location of a peer can be considered when deciding whether it should be or not a replica peer [83].

Request load balancing between replica peers. When an object is replicated at multiple locations in the system, the request load is shared between the owner and the corresponding replica peers. However, some replica peers may receive more requests than other replica peers holding the same object. To go even further with caching and replication, besides being shared, the load can also be balanced.

The Plaxton-type overlays that use the prefix match for replication have implicit load distribution between replica peers, but the load is balanced only when the requests come uniformly from the identifier space. However, this can be overcome by taking into consideration the flow of requests, as in [5].

From the perspective of the choice of the destination (replica peer) for a request, the request load can be balanced if the location of the replica peers are known [12]. In such case, a replica peer is selected and the request is sent towards it. The selection can be random (however not generally preferred, since having more information than just the location of a replica peer is easy to get and does not add significant costs) or based on some constraint. When the constraint is to balance the load, the system is leveraged (which means that the additional costs are rather low), as in [70], where the destination for a lookup request is chosen from the corresponding replica peers with a probability proportional to the maximum number of requests that the replica peer can support per time unit, information which is advertised by each peer. Similarly, the replica peer is chosen at random with a probability proportional to peer capacity (here, its availability to deliver the object) [83] from physically close nodes that form a cluster. The scheme is based on a network of super-nodes (high capacity nodes), each one responsible for a cluster, where an object is replicated only inside a cluster. In

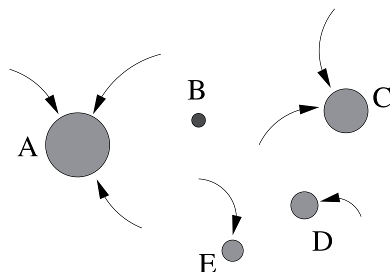


Figure 3.4: Example of replica peer selection: for request load distribution, the peers with higher capacity receive more requests.

order to determine to which cluster a node should belong to, distances are computed to a set of landmark nodes. The super-nodes gather load and capacity information from their cluster nodes, for both the replica peer selection and replica requests processing from overloaded nodes.

An example of replica peer selection based on their capacity is shown in Figure 3.4. The replica peers are represented with filled circles of size proportional to their available capacity. Their incoming requests are shown with curved lines. As can be seen from the figure, peer A, which has advertised a higher capacity, receives more requests than the rest of peers that have declared a lower capacity.

3.4 Traffic Routing

By object placement, we have shown how to share the objects between several peers, taking into consideration their number, size and request rate. We now go further and take in consideration the overlay paths that the lookup requests follow from their sources to their destinations, in order to deal with the routing load.

Even under a uniform flow of requests, the routing tables and strategies might use some overlay paths more than others for the traffic routing, hence overloading the nodes on these paths, while other nodes from the system might be hardly used, as noted in [78]. This imbalance gets much worse under popular requests.

Giving the fact that the traffic routing is determined by the current links (i.e., neighbors) and the routing strategy, we classify the routing load balancing solutions in the following two categories:

- *link reorganization*, where the balance is done by updating the overlay links, while the routing strategy remains unchanged;
- *path redundancy*, where the balance is done by the routing strategy, while the overlay links are the same.

Both types of solutions have the advantage of adapting to load changes.

To forward requests to other nodes, each node has a *routing table* that contains several other nodes from the system. Their number gives the *degree* of the overlay. An overlay with *constant* node degree has low maintenance costs for keeping live nodes at each entry in the routing table, but the choice for each entry is very small. Examples include de Bruijn-based overlays [41], Viceroy [57] or CAN [67]. Other DHTs use a *logarithmic* node degree, such as Chord [85], Pastry [71], Tapestry [97] or Kademlia [59]. These overlays show higher costs for maintaining the routing tables compared to the overlays that use a constant node degree, since usually their routing tables contain more entries. Nevertheless, they can use alternative entries when forwarding a request, which is a good start base for path redundancy. Moreover, some of them allow flexibility in the routing table entries, which is used in link reorganization.

3.4.1 Link Reorganization

In the logarithmic degree overlays, the choice for the neighbors is based on some strict [41, 57, 67, 85] or flexible [59, 71, 97] rules. For the latter, multiple nodes are suitable for each routing table entry, which has the advantage of changing the neighbors without affecting the routing strategy, and more importantly, without affecting the average path length of the requests that pass through it. The routing tables are updated even if the nodes that are currently in the routing tables are still alive. In eQuus [55], the routing tables are periodically updated with new nodes, however the main purpose is to assure that the nodes from the routing tables are most of the time alive. Next, we will present how a routing load balancing strategy benefits from this flexibility.

Pastry [71] is one of the overlays that allow flexibility in the choice of the neighbors. At entry i of a Pastry routing table, a node and the corresponding neighbor share the first i digits of their IDs. A first deduction is that for long-range neighbors (small i), the choice is much bigger, having much more suitable nodes at these entries. For example,

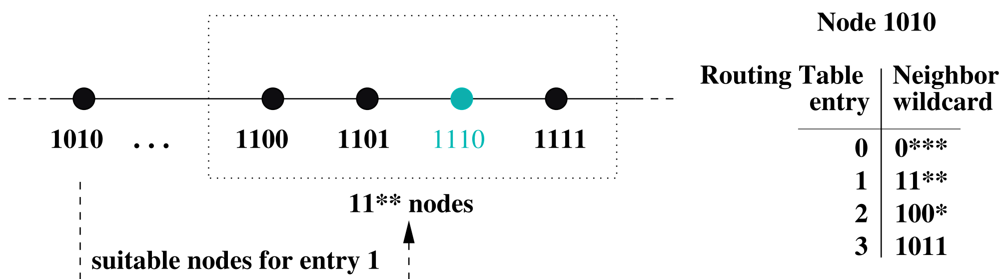


Figure 3.5: Flexibility in the choice of neighbors.

a node with ID 1010 in an identifier space of 2^4, with each digit represented by a bit, has $2^{(3-i)}$ possible node IDs for its i entry. However, not any of these IDs refers to a node. Figure 3.5 shows the wildcards for the routing table entries of node 1010 and, on an identifier space depicted as a line, the nodes that are suitable for its entry 1. The identifier 1110 does not name a node, thus the choice of node 1010 for its neighbor at entry 1 is from nodes 1100, 1101 and 1111. The neighbor can be chosen according to any predefined rule or condition.

In [75], presented in the context of a Pastry overlay to benefit from its flexible choice of the routing table entries, the idea of the load balancing solution is to update the routing tables while running the lookup requests. The load to be balanced is the aggregate of the routing load and the request load, which are considered as equal: receiving a request to forward it further is roughly equal to receiving a request to serve it (deliver it). The balance is performed through routing table updates, while adapting to experienced load conditions: high-loaded neighbors are replaced by low-loaded nodes. In order to propagate load information, each node keeps track of its own load and it piggy-backs its ID and load on the lookup requests. When a node receives a lookup request, it compares each received load with the load of its current node at the corresponding routing table entry, and if lower, it updates the entry. To keep up with the load variations of the neighbors, some optimizations are applied, such as load updates or estimations. The load balancing is thus done by decreasing the load of high loaded nodes and implicitly increasing the load of low loaded nodes. A node with a high load will receive less forwarding requests after being removed from other nodes routing tables. Its load cannot decrease lower than its request load, unless replication mechanisms are applied [5].

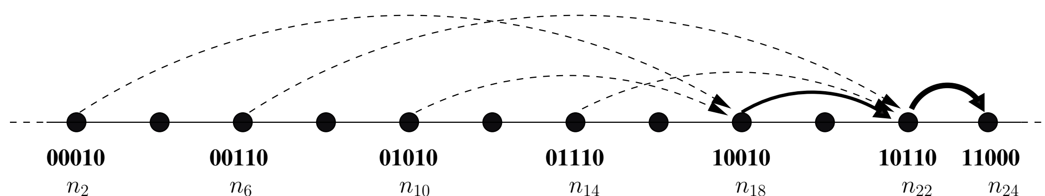


Figure 3.6: Example of path convergence towards the same destination, node n_{24}.

3.4.2 Path Redundancy

Path redundancy is related to the *routing strategy*, which decides the neighbor to be used as a next hop when forwarding a request. An overlay can have one or several routing strategies.

3.4.2.1 Single routing strategy

One of the most widely preferred routing strategies is incontestably greedy routing (employed for example in [71, 85]. With greedy routing, each request is sent at each hop as closest as possible to the destination, achieving a short path length under no failures. However, greedy routing has the drawback that the requests from various sources meant for the same destination will overload the nodes in the vicinity of the destination node, since they pass through the same nodes as last hops, confluence which is widely called *path convergence*. This kind of scenario usually happens when the selection of the neighbor to be used depends only on the requested key, i.e., at a node, the same neighbor is used for all requests for the same key.

An example of path convergence is shown in Figure 3.6, where several nodes (n_2, n_6, n_{10} and n_{14}) issue requests towards node n_{24}, on an identifier space of 2^5. The requests from n_2 and n_{10} arrive at node n_{18}, from where they start following the same path, passing through n_{22} in order to arrive at n_{24}. Additionally, at node n_{22}, this path joins with the path of the other two requests that are coming from n_6 and n_{14}. The first hop of each request is shown with a dashed line, while the path followed by two or more requests is shown with a filled line, being thicker when the path convergence is higher. The figure depicts node n_{18} and especially node n_{22} as being the mostly loaded with the forwarding traffic for destination n_{24}.

When only one routing strategy is used, the routing load can be balanced only if the routing strategy allows for flexibility in the choice of the neighbor to be used as next

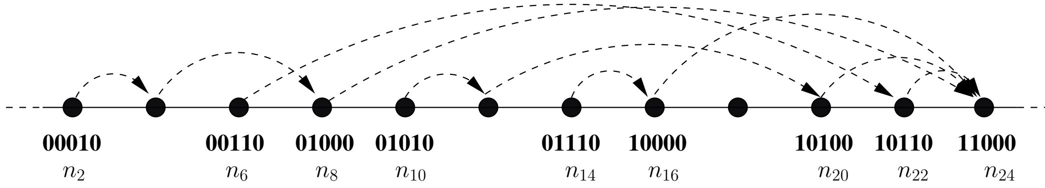


Figure 3.7: Load balancing on the forwarding traffic towards the same destination, node n_{24}, using random choice for the next hop.

hop for a request. The request travelling in the overlay would have alternative paths to follow. These kind of overlays usually have hypercube-like structures [2, 55, 73, 77], which provide alternative paths under similar path lengths.

HyperCuP [73] has a hypercube structure that is built as peers join the system. Each node keeps its neighbors on a per-dimension basis, and it might have the same node as neighbor in two or more dimensions (i.e., acting as several nodes), if no other suitable node has been found. When joining the system, a new node contacts an existing random node and becomes its new neighbor in its lowest free dimension. This random node is also responsible for providing nodes to the newly joining node for filling-in the rest of its dimensions. The strongest point of this solution is the idea of the hypercube construction, however a node acting as several nodes might limit the load balancing solution, because of the supplementary load it has to support.

The HyPeer overlay [77] (further detailed in Chapter 4) also adapts the hypercube as peers join and leave, however its construction is rather different. Each node has neighbors at exponential distances and the ones at exactly a power of 2 away are called aligned neighbors. The idea is to have as many aligned neighbors as possible. In order to achieve this, each joining node contacts a random node and becomes its new neighbor in its highest dimension that does not contain an aligned neighbor. This joining procedure has also the advantage of namespace balancing. When routing, the distance on the identifier space between the source and the destination is computed as a sum of hops, where each hop is a different power of 2. These hops can be taken in any order. Using randomness in the choice of the hop to follow in the request path is notably balancing the routing load in the system.

An example of utilization of such mechanism based on randomness is shown in Figure 3.7, which comes as a possible solution to the problem illustrated in Figure 3.6.

The same four nodes issue requests that arrive at node n_{24}. This time they take different paths (using some other routing strategy then greedy routing) and they do not converge, which balances the routing load.

In [2], Alvarez *et al.* propose to increase the number of path connections through the use of a hypercube structure. Each node has an identifier and a mask that indicates the identifier space that the node is responsible for. The routing algorithm can be either proactive, assuring a specific route to each node based on a tree distribution of the IDs, or reactive by creating on demand a route and keep it for a certain period of time. This usage of the hypercube could balance the routing load only if the number of generated routes is high.

Not only the structures based on the hypercube can offer alternative paths, most of the structured overlays can do this, but usually with an increased path length. As a consequence, they are mostly used for fault tolerance, where a longer path length is broadly accepted.

3.4.2.2 Multiple routing strategies

If the structure supports it, there might be several routing strategies to route requests. This is mostly the case for multiple-dimension structures such as CAN [67] or again, the hypercube-like structures. Multiple routing strategies for the same overlay are proposed for HyPeer, where each routing strategy has its own goal: short path length, load balancing, fault tolerance or short delay.

Even if no specific routing strategy is meant for balancing the routing load, choosing randomly the routing strategy to be used, either at the initiator node or, if possible, at each hop, would intuitively balance the routing load. Different routing strategies choose different neighbors as next hop for the requests with the same destination key, hence these requests would follow different paths. To our knowledge, this option has not been explored so far.

3.5 Load Balancing in the Underlay

There is not always a good correspondence between the overlay structure and the underlay topology, neither for the nodes to their physical location nor for the links to

the underlying route. It follows that the load balancing mechanisms applied on the overlay do not necessarily balance the load also at the underlay level.

In this context, the mostly considered issue is the *network traffic* and how can it be balanced and minimized. Even if the routing load is balanced at the overlay level, the network traffic might be sent only on a small set of underlying paths, charging more the corresponding resources. Moreover, even if a request has a short path length at the overlay level, at the underlying level it might zig-zag between opposite sides of the network, which would generate a large network traffic.

Balancing and minimizing the network traffic can be achieved by building the overlays with some knowledge about the underlying topology and then routing closer to the underlying topology structure while applying the load balancing mechanisms of the overlay. It is worth noting that these mechanisms are limited by the underlying topology itself: a stub network connected through a single link to the rest of the network does not have any other choice than following that single link for connectivity.

There are three globally known approaches to exploit network proximity [11]:

- *topology-based node ID assignment*: map the overlay nodes onto the underlay topology by assigning topology aware IDs;
- *proximity neighbor selection*: map the overlay links onto the underlay topology by selecting the physically closest nodes for the routing tables;
- *proximity routing*: route towards the physically closest suitable neighbor.

The first two approaches deal with the overlay structure in order to achieve a sound mapping of the nodes and the links onto the underlay topology. Then, the network traffic travels close to the underlying topology regardless the routing strategy that is used. When neither of the two approaches is taken into account, the only choice is proximity routing.

Topology-based node ID assignment. The nodes get identifiers that contain a certain information about their physical location in the underlay. The overlay can use it for either proximity neighbor selection or proximity routing.

The node ID assignment in [83] is based on Hilbert numbers, which assures that two nodes in the overlay are also close in the underlay network. Additionally, Hilbert

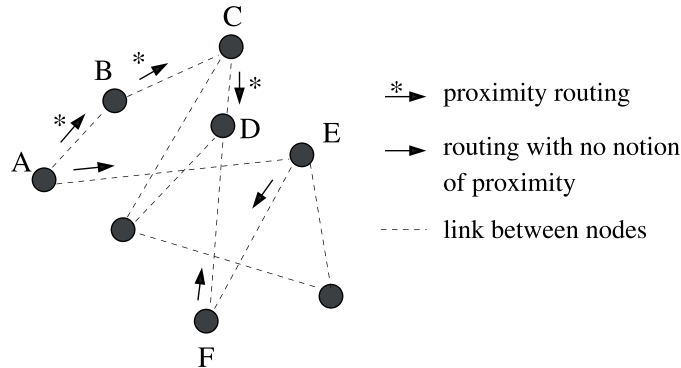


Figure 3.8: Request paths from A to D, with and without proximity routing.

numbers are also used to reduce the network traffic by properly choosing the two peers to perform a virtual server transfer [82, 98].

In TOPLUS [23], the proximity aware assignment is more relaxed, by assigning group IDs instead of node IDs. Proximity-close peers are hierarchically grouped in clusters, according to network IP prefixes. The routing is then done according to the obtained hierarchy.

Canary [51] uses network coordinates to assign identifiers to nodes. The network coordinates of the nodes are mapped to a CAN structure in order to integrate the physical link latency in the overlay. The zone of each node is adjusted according to the movement of its network coordinate.

Proximity neighbor selection. This approach is used in the overlays that allow for flexibility in the choice of the neighbors. The proximity-closest nodes are selected as neighbors, using a metric such as the number of IP routing hops or the geographic distance.

As we have seen earlier in the explanations of Figure 3.5 for a Pastry overlay, only for the long-range neighbors the choice is high. These routing table entries benefit the most from this approach.

Proximity routing. In an attempt to minimize the network traffic even when the overlay is not mapped on the underlay, proximity routing chooses as next hop for a lookup request the physically closest neighbor to itself among all suitable neighbors, as in [67] or [77].

An example of proximity routing is shown in Figure 3.8, where we have depicted at the underlay level the physical positions of some nodes. The links between the nodes are shown with dashed lines, while the lookup paths are shown with little arrows. In this example, node A sends two requests to node D, one with and another one without proximity routing. At node A, the requests can be sent through either neighbors B or E (we consider both of them suitable since they are closer in the ID space to the destination key than A). Node B is closer according to the proximity metric, but only the proximity routing strategy is aware of this. The other routing strategy blindly chooses node E. The decision for the neighbor to use is taken at each step. Finally, both paths have the same number of hops, however, the path that uses proximity routing has the advantage that is much shorter on the underlying topology, avoiding needlessly passing through the same routers (not shown in the figure).

The most convenient way for a node to detect which nodes are close (either for proximity neighbor selection or for proximity routing) is to probe them, however this generates too much network traffic when the probe is done periodically. A lighter solution is the usage of network coordinates [13], where each node has coordinates in a multi-dimensional space, and the distance between two nodes is estimated as the euclidean distance between their coordinates. The coordinates of a node are periodically updated when a reply to a previously lookup request that it had launched is received, which carries also the coordinates of the destination node. The computations take into consideration the real distance to the destination node (computed locally), the current estimation of this distance and the coordinates of the destination node. While the network coordinates use estimations, the interaction between the overlay and the underlay can use predictions, as in [74], where for proximity routing the next hop is chosen as the node that is on the other end of the most preferred link. Alternatively, the ISP can provide information related to the proximity of the peers [1], where, given a list of peers, the ISP replies with the list sorted according to some proximity metric.

In order to route closer to the underlying topology while achieving namespace balancing, *Nikolaos et al.* [17] use two CAN overlays: one that is built according to a load balancing algorithm for namespace balancing (Virtual CAN), while the other one reflects the underlying network (Locality CAN). To minimize the network traffic, a

request is first routed in the Locality CAN and then redirected to the destination in the Virtual CAN.

The network traffic of the underlay is also balanced for data transfers, not only for the network traffic generated by the lookup requests. For content delivery, [32] places all the nodes in a hypercube network and reduces and balances the network traffic by assigning clients to appropriate servers, based on their capacity. For gossip dissemination, in the context of self-organized systems, [80] uses network-friendly protocols that rely on combinations of *push* and *pull* strategies for destination peer selection. Two views are used to provide close and random neighbors, respectively. The estimation of the distances between the nodes is based on network coordinates.

3.6 Summary and Discussion

In this survey, we have identified the main causes of load imbalance in peer-to-peer systems. We have detected four contexts where load problems can occur: overlay namespace, requests, routing and underlying topology. In each context, the load is represented by a different metric: number of keys per node, request load, routing load and underlying traffic load, respectively. We have then classified the solutions for these load problems into three categories: object placement, traffic routing and underlay, and we have presented in detail the most relevant existing solutions, which we now shortly review through discussion.

Starting from early research papers on peer-to-peer systems, load balancing was considered as a primary concern next to security or fault tolerance issues. This is however not surprising since the aspects to be treated were mainly inspired from the world wide web. For example, the consistent hashing in [42] or the location awareness in [64], used initially in caching and replication mechanisms proposed for the world wide web, found their applicability later in peer-to-peer systems.

The advantage of caching and replication is that they are highly applicable, given the fact that they do not modify the existing overlay (structure or routing related), instead they add extra copies of the same object. Their configuration is based on the choice of objects to replicate, the choice of peers to use for replication and the choice of the destination peers for the lookup requests. Usually, periodic replication requests are done, because some of the objects might have been dropped, for example by the

cache replacement policy, in order to accommodate objects from new replica requests. The price of this kind of solution is the extra storage needed at some peers, but the complexity of the solutions is rather low. Making good choices, they have been shown to deliver good results [5, 12, 28, 34, 56, 65, 70, 81, 83, 91, 92, 94, 95].

The usage of multiple functions to hash an object and redirection pointers is a rather simple mechanism [10, 67]. The choice of the peer to store the object is the only thing that adds some complexity. Then, the choice of the destination(s) for the lookup requests is usually random. This solution only checks for load imbalance when an object is added into the system. So its drawback is that it would not re-balance the load if objects are removed.

Still related to the object placement, namespace balancing [6, 42, 47] and the usage of virtual servers [14, 26, 27, 43, 53, 66] are a bit more complex. Namespace balancing defines the way of assigning identifiers to nodes and objects, and it is critical to have a balanced namespace as a load balancing solution to start with. Except consistent hashing, the overlay joining schemes are rather complex in the sense that a node has to carefully choose the interval between two existing nodes where to get its identifier. This gives the first position of the nodes and the objects in the identifier space, which can be later modified by the usage of virtual servers, or by induced churn.

With virtual servers, a node has one or several positions on the identifier space, and it can bind a new position or unbind an existing position in order to increase or decrease, respectively, its corresponding object load. These operations can be done in a centralized or decentralized manner, in order to balance the load in the system. The good thing with virtual servers is that the transfers are done continuously, which means that they adapt to new system conditions, so their usage is also appropriate in systems that deal with nodes arrivals or departures, or object addition or removal.

Going into traffic load balancing, there are two techniques that are complementary: either by continuously changing the neighbors of each node while using the same routing strategy (link reorganization [55, 75]), or conversely, by keeping the same neighbors but continuously changing the routing strategy (path redundancy [2, 55, 73, 77]). With link reorganization, the routing strategy uses the *same link* (entry in the routing tables) to forward consecutive requests for the same destination, but due to the updates, this link changes the node that it points to, which makes the load being balanced on different nodes (current and former neighbors). With path redundancy, the routing strategy uses

different links to forward sequential requests for the same destination (the links are not updated with new neighbors unless they fail), which balances the load over the current neighbors. Both solutions balance the routing load, however using different input. Link reorganization deals with the load of the nodes, which means sending additional information for a node to know the load of another node, while path redundancy deals with the load of links, which means keeping track locally of the load it has generated on each of its links. Therefore, link reorganization offers a higher choice of neighbors, but path redundancy is lighter. To our knowledge, yet there is no proposed solution to cover both link reorganization and path redundancy.

The overlay traffic might generate too much traffic at the underlay level, thus solutions have also been proposed in this sense [1, 11, 13, 17, 23, 32, 67, 74, 80, 83]. The best solutions so far for reducing the load at the underlay level are to map the overlay as close as possible to the underlay and then let the underlay deal with traffic load balancing. This can be done by carefully choosing the neighbors of each node: with proximity neighbor selection, the neighbors that are close according to a predefined proximity are selected for the routing table. However, it is not always effortless or possible to do this mapping of nodes and links to the underlay. Where applicable (i.e., flexibility of the routing strategies), proximity routing is used to forward the requests, by choosing the closest suitable neighbor. In any case, topological information is needed from the system, either from central entities (e.g., ISP) or by extra messages or piggy-backed information (e.g., network coordinates) or topology-based node ID assignment. Knowing the IDs or the network coordinates of two nodes is enough information to give a rough estimation of the distance between them at the underlay level, which can be used either for proximity neighbor selection or proximity routing.

Our survey on load balancing tackled the load in object placement and routing traffic at the overlay and underlay level. Each of these load categories can be still explored, and moreover, new categories may span from new metrics of load.

Chapter 4

HyPeer: Structured Overlay with Path Redundancy

"If you want to succeed you should strike out on new paths, rather than travel the worn paths of accepted success."

John D. Rockefeller

4.1 Introduction

[1] The numerous and diverse designs of peer-to-peer distributed hash tables (DHTs) proposed in the literature propagate lookup requests along paths relying on routing strategies that consider mainly the node identifier values. As a consequence, many important system parameters and properties are not taken into account, which can affect the reachability of the information or the response time. When considering for instance network delays, it might be advantageous from the performance point of view to choose a longer, but faster path than the one provided by a purely identifier based routing strategy as it is for example the case in Chord-like systems [85]. To address the shortcomings stemming from these system parameters and their probable dynamic changes, we propose a new peer-to-peer DHT overlay structure, HyPeer, that allows the existence of different routing strategies (either identifier based or not) making a different choice of path. The objectives of the routing strategies can be various; here, we consider fault tolerance, load balance and low latency.

[1] The HyPeer overlay was presented at the DSN DASSON [78] workshop and at the CoopIS [79] conference. An extended version of this chapter has been submitted to Elsevier ComNet [77].

To achieve such flexibility, the overlay structure must be organized in a way that provides *alternative paths*, allowing for strategies featuring a “flexible-choice routing” property. The hypercube structure offers several disjoint paths between any two nodes and permits therefore routing strategies that exploit alternative paths in the structure in order to achieve the different objectives stated above. These observations motivate the design of the HyPeer overlay, which is loosely based on a hypercube structure. Experimental evaluation has indeed shown that this simple peer-to-peer overlay structure allows us to reach the objectives of fault tolerance, load balance and low latency under a large variety of scenarios taking into account churn, node failures and various load and request distributions, as well as network (delay) conditions. Moreover, the defined routing strategies on the hypercube approximated overlay structure are shown to keep the average path lengths short while improving the desired performance.

Alternative paths are essential in the case of failures. When the path to a destination contains dead nodes, another path can be used instead. In particular, alternative paths serve to route requests even before the structure has been repaired following the failure. They contribute therefore to increase *fault tolerance* in routing.

Furthermore, alternative paths are instrumental in more evenly distributing the flow of requests in the system. Indeed, links and nodes may suffer from overloading under very popular requests. By choosing alternative paths in this case, the routing load (number of requests a node has to forward towards their destinations) can be more evenly distributed over the system. This provides *routing load balancing* over the nodes and links of the overlay.

While a greedy routing strategy seeking to approach the destination as much as possible in each step results in short path lengths, such paths are not necessarily the most efficient ones. Indeed the corresponding path in the underlay network may take a longer time to traverse. It is possible that using a different routing strategy (i.e., choosing a path of longer length) may be more efficient because it might show lower network delays (or latency). A *proximity routing* strategy could use network delay as proximity metric between nodes when choosing from a set of alternative paths. Such a strategy is expected to yield on average lower routing path delays and therefor increased performance.

The HyPeer DHT overlay that we propose has a logarithmic node degree structure similar to Chord, but assigns identifiers in a way to better approximate a hypercube

structure. This design allows for routing strategies that exploit alternative paths providing flexible-choice routing according to the objectives of fault-tolerance, load balance, and low latency.

The next section presents the background and the motivation of our work, and discusses the flexible-choice routing advantages. In Section 4.3 we present the structure of our system and the four routing strategies, *Greedy routing, Fault-tolerant routing, Load-balanced routing, and Proximity routing* that leverage from the alternative paths choice property. The system and its routing strategies are experimentally evaluated in Section 4.4. Finally, we conclude in Section 4.5.

4.2 Background and Motivation

In our design, we use an overlay structure loosely based on a hypercube because of the flexible-choice routing property it offers: indeed, there are several disjoint paths between every two nodes of a hypercube. This desirable property may obviously not hold if the structure is not fully populated, e.g., because some of the vertices are missing due to peer departures or failures (churn), or in order to support peer arrivals. Therefore, we will use overlay structures that approximate hypercubes and provide similar properties most of the time.

To illustrate the principles underlying our approach, let us first consider the well-known Chord [85] DHT that we will generalize for HyPeer. In Chord, nodes receive a random identifier in the range $[0; 2^m)$ and they are organized in a ring structure based on these identifiers. In addition to its predecessor and successor on the ring, each node keeps track of other nodes located at exponentially increasing distances from itself. Specifically, each node has a reference to the first other node encountered on the ring at a distance of *at least* 2^i clockwise, with $i < m$. Therefore, the degree of the network is $O(\log N)$ given a population of N nodes.

A node in the network is responsible for all identifiers—also called *keys*—located between its predecessor (excluded) and itself (included). Requests are routed using a greedy algorithm: when looking up a given key k, each node n on the route forwards the request to its neighbor located between n and k that is closest to k, if any, or to its successor. This process stops when reaching the node responsible for k. Therefore, a request routed from a source to its destination will take steps of length $2^i + \varepsilon$, where ε

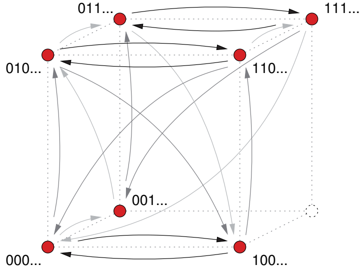


Figure 4.1: Representation of HyPeer as an approximation of a hypercube.

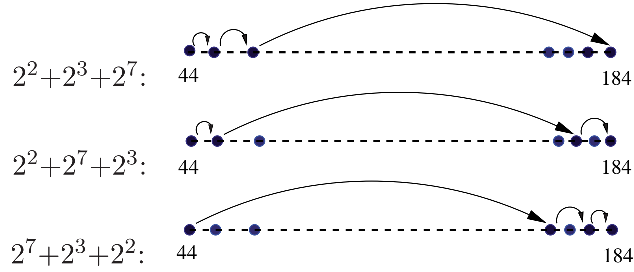


Figure 4.2: Several different paths can be used to cover a given distance (140 here) between the source and the destination nodes.

is a value that depends on the placement of the nodes on the ring and i decreases at every routing step (greedy routing).

HyPeer follows the same underlying principle: each node in the overlay has a set of m neighbors at exponential distances, but it additionally adds accuracy in the neighbors' positions, i.e., it controls the placement of the nodes on the structure such that any two nodes are placed with high likelihood at a distance of *exactly* 2^i (see the hypercube representation in Figure 4.1). This way, the requests can take steps of length 2^i in any order (the resulting paths having under ideal conditions the same length) rather than relying only on greedy routing, only. This opens way to alternative routing strategies.

Considering Figure 4.2, one observes that the distance of length 140 from node 44 to node 184 can be covered in three hops of 2^2, 2^3 and 2^7 in any order, since their sum is 140 and addition is commutative. The number of possible paths corresponds to the permutations of these hops, which in this case is 6 distinct paths (3!). Three out of the six possible paths are shown in the figure.

Definition. We call an *aligned neighbor* a node whose identifier is at a distance of exactly 2^i from the current node. An *aligned link* is a link that points to an aligned neighbor.

As in the previous example, in order to compute the number of possible paths, we first express the distance between the source and the destination as a sum of exponents of our base, i.e., 2. Then, the number of terms gives the number of hops, that we denote as h. Even if not all the neighbors are aligned (e.g., because of churn), we can

use as a rough estimation of the number of possible paths the value $h!$ (permutations of h). After a hop of any length, the number of remaining paths of size $h-1$ is $(h-1)!$. Note that this does not depend on the length of the hop taken. However, the number of alternative paths of more than $h-1$ hops that can be used to route around failures is higher when a hop of smaller length has been used, since the identifier space left between the current node and the destination is larger. We also note that, when taking a long hop, we typically pass over some nodes with direct links to the destination; this has implications for fault tolerance as we will discuss in Section 4.3.2.

Figure 4.3 shows an example where source node n_2 sends two requests to destination n_{24}, one with a greedy routing strategy (marked with GR) and the other with a fault-tolerant routing strategy (marked with FT). If there are no failures, the destination can be reached in three hops of lengths 2^1, 2^2, and 2^4. The GR strategy tries to send the request as close to the destination as possible, thus taking the longest possible hop first. In contrast, the FT strategy proceeds cautiously and sends the request through the shortest hop first in order to keep as many alternative paths as possible to better route around failures. The paths of the two requests are shown with solid lines, while the nodes that have direct links to the destination node n_{24} (i.e., its incoming links) are shown with dotted lines.

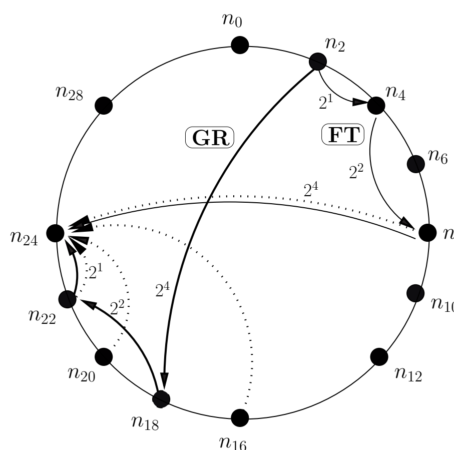


Figure 4.3: The request path to be followed by two routing strategies (filled lines) from source node n_2 to destination node n_{24}. The incoming links of node n_{24} are marked with dotted lines.

To observe the behavior of the two routing strategies when failures occur, let us assume that some of the incoming links are failing. We first consider that nodes n_{20} and n_{22} are dead. The GR request arrives from n_2 to n_{18}, where it cannot proceed any further: in its attempt to forward the request as close as possible to the destination, the GR routing algorithm discards other incoming links at the destination (live nodes

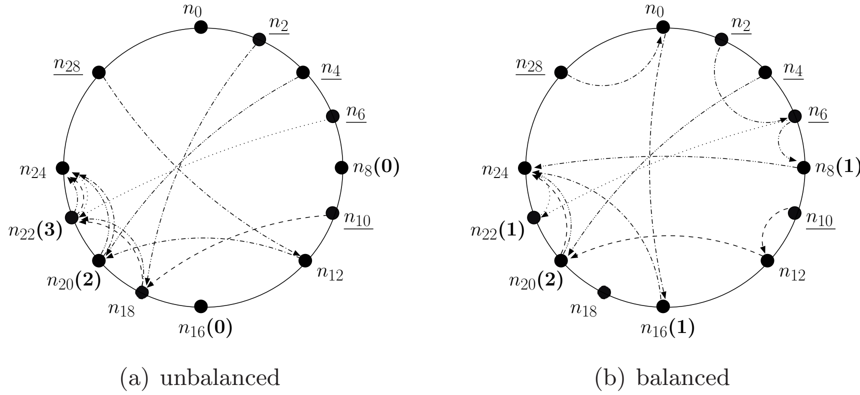


Figure 4.4: Traffic on incoming links of node n_{24}.

n_8 and n_{16}). While this problem can be dealt with by using backtracking, this comes with an additional cost. In contrast, the FT request is routed through nodes n_4 and n_8 before being forwarded directly to the destination via a long hop.

Let us now consider that nodes n_{20} and n_{22} are alive, but n_8 and n_{16} are down. Now, the GR follows its usual path (the hops 2^4, 2^2, and 2^1 in this order), while the FT request must skip dead nodes and pass via nodes n_6 and then n_{22}. In both scenarios, FT routing succeeds despite dead nodes and without need for backtracking, while traditional GR routing fails in the first case even though there exist several alternative paths between the source and the destination.

Another important property of flexible-choice routing is its potential for balancing the traffic load of a popular node among multiple paths. To illustrate the principle of load balancing, let us consider again the same sample overlay, but in a scenario where five nodes send a request to node n_{24} (e.g., responsible for a popular key) and where there are no failures. We depict the requests following two routing strategies in Figure 4.4: (a) a greedy routing strategy that takes hops from the longest to the shortest and (b) a routing strategy that takes hops in some random order. The source nodes are underlined and their requests are shown with different types of lines.

The number of requests that each incoming link forwards in a direct hop to the destination is shown in parentheses after the node identifiers. We observe that the number of forwarded requests per incoming link, starting with the furthest ones, is 0, 0, 2, 3 in 4.4(a), while it is 1, 1, 2, 1 in 4.4(b). Albeit the sum being identical,

GR exhibits an unequal distribution of the load over the incoming links, while the distribution is much more balanced in the case of randomized paths.

To sum up, flexible-choice routing offers promising capabilities for fault tolerance and load balancing. In the next section, we describe the HyPeer overlay that has been specifically designed to take advantage of these capabilities.

4.3 HyPeer Overlay

4.3.1 HyPeer Structure

The HyPeer structure spans over a space of 2^m identifiers (where m is the number of bits for each identifier). As Chord, it organizes nodes in a ring. Each peer also maintains a routing table RT with $O(\log N)$ entries ("fingers") pointing to neighbors at exponentially increasing distance. N is the number of nodes in the system, and finger i designates the first peer on the ring located at distance of at least 2^i clockwise from the current node.

Unlike Chord, HyPeer assigns identifiers and selects remote neighbors in a way that the resulting structure approximates a hypercube of dimension $\log N$. Ideally, nodes in HyPeer should only have aligned neighbors and the inter-node distance on the ring should be constant. In other words, given a population of $N = 2^x$, every node should have x neighbors at distance $2^{m-1}, 2^{m-2}, \ldots, 2^{m-x}$. However, obtaining such an optimal configuration is unrealistic in the context of peer-to-peer systems, where nodes join and leave at any time. This means that a node may have some neighbors that are not aligned, and the number of neighbors may vary slightly among the nodes. Our contribution, when comparing with Chord, is to significantly increase the chances for the structure to have aligned links in order to adopt new flexible routing strategies.

To control the placement of the nodes in the identifier space, we do not choose random uniform identifiers for the nodes as most other DHTs do (typically by hashing the IP address of the node); instead we delegate to an existing node n the task of assigning an identifier for a joining node n_j. Node n searches in its routing table an entry where there is no aligned neighbor, and assigns the corresponding identifier to n_j. We will discuss refinements of this assignment strategy later in this section.

The simplest way for choosing node n is to pick it at random, e.g., by looking up a random key. Alternatively, node n can be chosen to meet a specific criteria. A typical

example would be for the new node to select a nearby node, e.g., in the same AS or ISP. This may help reduce the latency and the stress on the communication links, as nodes that are close in the underlying network topology will be neighbors in the DHT. This may though also lead to imbalances in the structure and degrade the quality of the hypercube approximation. For simplicity, in our evaluations, we will consider n as randomly chosen.

Algorithm 4.1 JOIN(n_j) at node n

```
1:  if dist(pred(n), n) > dist(n, succ(n)) then
2:      Send JOIN(n_j) to pred(n)
3:  else
4:      {Highest entry x without aligned neighbor}
5:      x ← argmax_i RT[i] s.t. ¬ALIGNED(n, RT[i])
6:      if x ≠ ⊥ then
7:          n_j ← n + 2^x
8:          RT[x] ← n_j
9:          return n_j
10:     else
11:         Send JOIN(n_j) to pred(n)
12:     end if
13: end if
```

The joining procedure that we use to build the hypercube-like structure is shown in Algorithm 4.1, where we consider that the joining request has already arrived at n. The function returns the free identifier to be assigned to the joining node. This identifier is chosen to be at a distance of exactly 2^i in order to increase the number of aligned neighbors of node n. To simplify presentation, we assume that function ALIGNED(p, q) returns true iff q is a live aligned neighbor of p, i.e., the distance between both peers is a power of 2.

We apply two optimizations to this basic mechanism. First, in order to place nodes evenly on the ring, i.e., with an inter-node distance that does not vary too much, the predecessor of n is requested to assign the new identifier if it has a larger inter-node distance with its successor than n (lines 1–3). Second, in order to fill the hypercube structure per dimension, from the highest to the lowest, n chooses the free identifier in its routing table that is the furthest away (lines 4–5). Node n updates entry $RT(x)$, which points to the neighbor at entry x of the routing table, with the joining node n_j

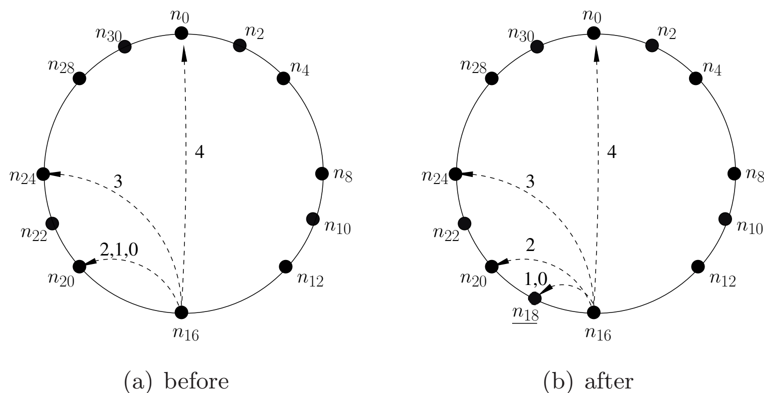


Figure 4.5: Aligned neighbors of node n_{16} when a new node joins the overlay.

and returns the new identifier (lines 6–9). If node n already has only aligned neighbors, the request is sent to its predecessor (lines 10–12).

There are various alternatives to the condition at line 1. For example a joining request could be sent to the predecessor of n if the latter has fewer aligned neighbors than n. This would better balance the number of aligned neighbors per node at the price of extra communications between nodes during the joining procedure. In our experiments we only use the default condition as shown in Algorithm 4.1.

An example of a node joining the overlay is presented in Figure 4.5 on an identifier space of $2^{m=5}$, starting at the randomly chosen node n_{16}. The distance from n_{16} to its predecessor is not smaller than the distance to its successor, thus n_{16} assigns the new identifier. It checks its aligned neighbors, shown with dashed lines in Figure 4.5(a), starting from the highest entries. The highest entry of its routing table that has not an aligned neighbor is at index 1, which means that the new node will get the identifier $16+2^1 = 18$. After joining, node n_{16} has 4 aligned neighbors, as shown in Figure 4.5(b).

The joining node must be added to the routing tables of the other participating nodes. To that end, we rely on a periodic check made by every node to detect whether its aligned neighbors are still alive and whether some non-aligned neighbors should be replaced in its routing table by another node, likely a newly joined aligned node.

To evaluate the quality of the structure constructed by HyPeer, in our evaluations we will compare it against an ideal structure where all nodes have the same number of aligned neighbors placed on the same dimensions (i.e., inter-node distances are identical across the ring). This ideal structure determines the best setting for evaluating our

routing strategies. We thus define FH, a full d-dimensional HyPeer as the ideal case of a HyPeer structure with 2^d nodes. All nodes have d aligned neighbors at the highest entries of their routing tables and an inter-node distance of $2^{(m-d)}$.

4.3.2 HyPeer Routing Strategies

In HyPeer, like in Chord, a key is under the responsibility of the first node that has an identifier greater than or equal to that key. We have designed the HyPeer structure to support several routing strategies for looking up the node responsible for a given key, from plain greedy routing to specific routing strategies that take advantage of the aligned neighbors. Requests always traverse the ring clockwise by following neighbor links.

Strategy	Short Description
Gr-HyPeer	Use furthest aligned neighbor Goal: *Short routing path*
FT-HyPeer	Use closest aligned neighbor Goal: *Fault tolerance*
LB-HyPeer	Use random aligned neighbor Goal: *Load balancing*
Pr-HyPeer	Use lowest-delay aligned neighbor Goal: *Proximity routing*

Table 4.1: Routing Strategies in HyPeer.

We define four HyPeer-specific routing strategies that are summarized in Table 4.1. They all make use of aligned neighbors. A node computes the remaining distance to the destination as the sum of hops with distance equal to a power of 2, and chooses one of the aligned neighbors found at such exponential distance. We denote such aligned neighbors as “eligible”. When the request is forwarded to an eligible neighbor, the estimated distance to the destination decreases by a power of 2, and the expected number of hops by one.

Greedy routing. Gr-HyPeer uses the furthest eligible aligned neighbors. This corresponds to a greedy routing strategy where the request takes long steps initially, then shorter ones when approaching the destination. As there are on expectation more

aligned neighbors on higher dimensions, and GR-HYPEER uses these neighbors first, this strategy is expected to provide the shortest average routing path.

Fault-tolerant routing. FT-HYPEER selects the closest eligible aligned neighbor. In contrast to GR-HYPEER, the request takes short steps in the beginning, then longer ones. It follows that, after each hop, the identifier space between the current node and the destination remains larger than for GR-HYPEER; this leaves more routing paths open and increases fault tolerance.

Load-balancing routing. The aim of LB-HYPEER is to balance the *routing load* (i.e., the load induced by traffic forwarding) on outgoing links to compensate for heavy biases in the request distribution. Indeed, when some keys are requested much more often than others, the load on the routing paths to such popular keys is much higher than on paths to rarely requested keys. We implement load balancing by simply choosing an eligible aligned neighbor at random.

Proximity routing. PR-HYPEER is a routing strategy that uses the network delay (or latency) between peers as metric.[1] From all the eligible aligned neighbors, PR-HYPEER chooses the one that is the closest in terms of delay. The choice of the neighbor to forward the request to is a local decision, thus the total delay of a multi-hop request may be suboptimal. However, this strategy is expected to yield *on average* a lower routing path (end-to-end) delay than the other routing strategies.

Algorithm 4.2 shows the selection at a node n_a of the aligned neighbor to be used as next hop for a request, according to the routing strategy.

The function NEXTALIGNEDHOP takes as parameters the HYPEER routing strategy rs applied for the request and the requested key k; it returns the aligned neighbor to which the request should be forwarded. If no aligned neighbor exists or if they are all dead, $\perp$ is returned.

Definition. We define χ as the list of values such that $\sum_{i\in\chi} 2^i$ is the distance from the current node to the destination key. The cardinal of χ is the estimation of the number of hops towards the destination, regardless of the routing strategy.

[1] One might use other proximity metrics, such as the number of IP routing hops.

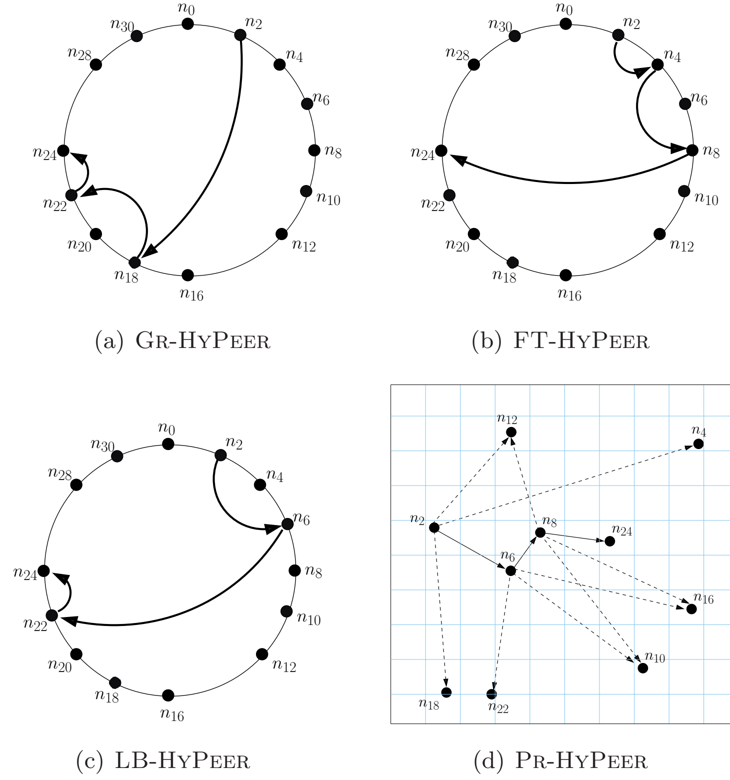
Algorithm 4.2 NEXTALIGNEDHOP(rs, k) at node n

```
1: if rs = GR-HYPEER then
2:    SORT(χ, DESCENDING)
3: else if rs = FT-HYPEER then
4:    SORT(χ, ASCENDING)
5: else if rs = LB-HYPEER then
6:    SORT(χ, RANDOM)
7: else if rs = PR-HYPEER then
8:    SORT(χ, DELAY_ASCENDING)
9: end if
10: x ← argmin_i χ[i] s.t. ALIGNED(n, RT[i])
11: if x ≠ ⊥ then
12:    return χ[x]
13: end if
14: return ⊥
```

We first sort the list χ according to the routing strategy. GR-HYPEER prefers longer hops, so it sorts the list in descending order (lines 1–2); FT-HYPEER prefers shorter hops so it uses ascending sort (lines 3–4); LB-HYPEER selects a random link and thus shuffles the array (lines 5–6); finally, PR-HYPEER sorts the array in ascending order of network delay (lines 7–8). After the ordering is done, the function selects the first entry of the list that corresponds to a live aligned neighbor of n (line 10). If such a peer exists, it is returned; otherwise the function returns $\perp$ (lines 11-14).

Lookup in HYPEER is implemented by repeatedly calling NEXTALIGNEDHOP at each node along the path. If the function returns $\perp$ (i.e., there is no eligible aligned node that is live), then we choose the furthest live node (not beyond the destination) from the routing table.

There are two important optimizations that we add to this protocol. First, nodes do not only store in their routing table the addresses of their neighbors, but also the key range under the responsibility of these neighbors (i.e., the distances from these nodes to their predecessors). These ranges are updated periodically by the self-stabilizing protocol that periodically verifies if neighbors are still alive. When routing a request, a node checks in its routing table whether the target key is under the responsibility of one of its neighbors; in that case, the request is directly forwarded to that neighbor. Note that information about ranges might be inaccurate, e.g., because the predecessor of a neighbor has changed, but this will not prevent the request to reach its destination.



(a) Gr-HyPeer (b) FT-HyPeer

(c) LB-HyPeer (d) Pr-HyPeer

Figure 4.6: Routing path from node n_2 to node n_{24} for each routing strategy.

In the worst case, the request will have to follow predecessor links when forwarded to a node whose key range was overestimated.

The second optimization helps avoiding pathological cases where requests follow very long paths. This happens in the unlikely situation when all eligible aligned links point towards nodes that are down, so the other links have to be used. After following a non-aligned link, the next node in the path recomputes the χ list, which contains some distances that are smaller then the distance covered in the last aligned hop. Therefore, and especially in the case of FT-HyPeer, the request would have to cover again small hops such as successor links, and if the situation persists, along a very long path. To avoid or alleviate this problem, when selecting the next hop from the list χ and when such a situation has already been detected, the successor links are not anymore taken into consideration for routing (unless there is no other choice).

An example of lookup with no failures is shown in Figure 4.6. Node n_2 looks up

key n_{24}. In (a), (b), (c) the nodes are depicted on the ring structure, while in (d) the nodes are depicted on an underlay level map in order to show their proximity. The path between n_2 and n_{24} is shown with solid lines. The dashed lines in (d) represent the other links (of the nodes on the path) that point towards eligible nodes.

The hops to be followed are of sizes 2^1, 2^2 and 2^4. These hops are taken in a different order, depending on the routing strategy. (a) Gr-HyPeer takes hops of decreasing length: 2^4, 2^2 and 2^1. (b) FT-HyPeer makes longer and longer steps. (c) LB-HyPeer chooses randomly, in this example 2^2, 2^4 and 2^1. (d) Pr-HyPeer sends the request to the closest neighbor in terms of delay, in this example following the path n_6, n_8, n_{24}.

In our evaluations, we use the same HyPeer routing strategy for all requests and along the whole routing path. However, the routing strategy could be chosen by the initiator node for each request. Moreover, a node could decide locally the routing strategy for the next hop, irrespective of the strategy used for the previous hop. Additionally, having support for alternative paths, new routing strategies can be driven by new objectives, or hybrids of existing routing strategies can be applied in order to achieve, however with less efficiency, different objectives.

4.4 Evaluation

In this section we present and discuss simulation results of HyPeer. We first focus on its structure and then evaluate the routing strategies. In our experiments, we use N=8,192 nodes that span over an identifier space of $2^{m=31}$. Nodes are added incrementally using the join algorithm of Section 4.3.

To evaluate the routing strategies, we inject 200,000 lookup requests in the system. The initiator and the requested keys are chosen uniformly at random in most experiments, except when observing load balancing. In that case, we perform experiments under 4 possible load-flows, which combine two selection types (random-uniform and Zipf-like) for the initiator node and the requested key (see Table 4.2). The Zipf-like selection has parameter α=1.0.

Intuitively, a Zipf-like selection of the initiator nodes is expected to generate requests that follow the same few paths in the beginning (since there is a small set of nodes performing most of the requests), while a Zipf-like selection of the requested keys is

Load Flow	Initiator nodes selection	Requested keys selection
uu	random-uniform	random-uniform
uz	random-uniform	Zipf
zu	Zipf	random-uniform
zz	Zipf	Zipf

Table 4.2: Load-flows: selection of initiator nodes and requested keys.

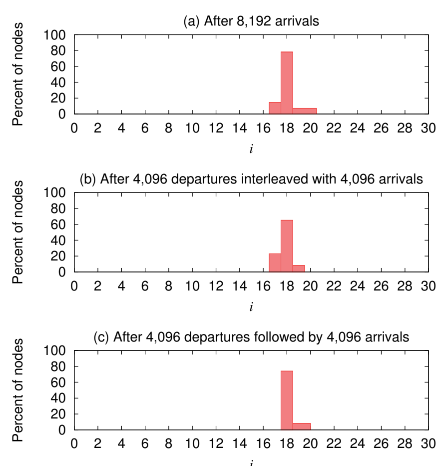


Figure 4.7: Percentages of nodes with an inter-node distance of 2^i.

expected to generate requests that follow the same few paths towards the end (since there is a small set of keys requested most of the time).

4.4.1 HyPeer Structure

The following evaluation of the HyPeer structure shows that it provides (1) a balanced inter-node distance and (2) a good placement of the nodes, i.e., most neighbors are aligned.

4.4.1.1 Inter-node distance

Figure 4.7 shows the inter-node distance of all nodes, as the percentage of nodes that have a given 2^i inter-node distance to their successor. After the $8,192$ nodes have joined the system (see Figure 4.7 (a)), almost 80% of them have an inter-node distance of 2^{18}, while the resting have very close values: 2^{17}, 2^{19} or 2^{20}. Such variations are expected

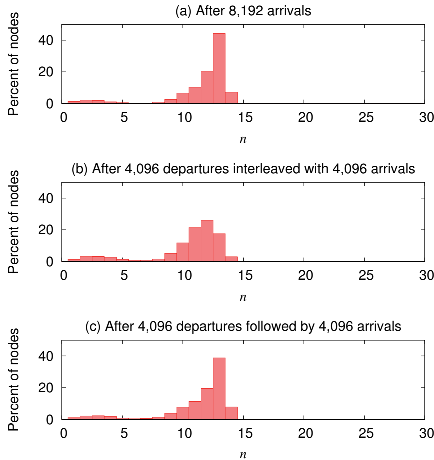


Figure 4.8: Percentages of nodes with n aligned neighbors.

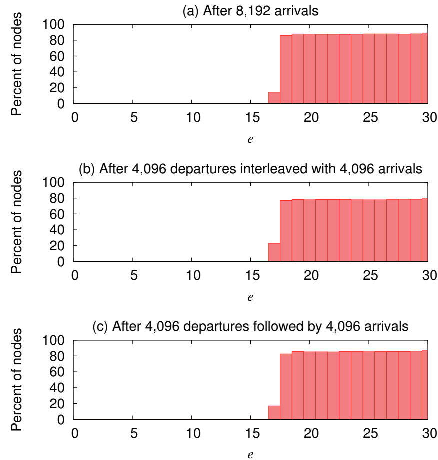


Figure 4.9: Percentages of nodes with an aligned neighbor at entry e.

to happen given the randomness of the join process and the continuous change in the structure caused by the arrival of new nodes. There is also a small fraction of nodes that have an inter-node distance that is not a power of 2 (not shown in the figure), but they represent only 0.05% of all nodes.

In order to evaluate the behavior of the structure under churn, and especially to cover a high range of realistic churn, after the 8,192 nodes have joined, we analyze two scenarios where half of the nodes leave and the same amount of nodes join: (1) a scenario where each departure is followed by an arrival and (2) a scenario where all the departures happen before the arrivals. The former is expected to produce a structure with more variance in the inter-node distances because it is less likely that a joining node will reuse the identifier of a departing node. In Figures 4.7 (b) and (c) we note that under both scenarios, churn affects only slightly the overlay structure as the inter-node distance remains almost unchanged. Additionally, the number of nodes with an inter-node distance different to 2^i remains small: 3.5% of nodes for (b) and only 0.07% for (c).

4.4.1.2 Aligned neighbors

We now analyze the number of aligned neighbors per node and their placement in the different dimensions of the hypercube.

According to the results presented above, the inter-node distance for most of the nodes is 2^{18}. Since the identifier space is 2^{31}, we can expect that a large number of nodes will have $31 - 18 = 13$ aligned neighbors. Moreover, the minimum inter-node distance is 2^{17}, thus we cannot expect to have more than $31 - 17 = 14$ aligned neighbors per node. Figure 4.8, which shows the percentage of nodes that have a certain number of neighbors under the scenarios considered, supports these conjectures. It also shows that only few nodes have less than 10 aligned neighbors, while most of the nodes have a large number of aligned neighbors, which is key for flexible-choice routing.

Likewise to the inter-node distance analysis, we observe that churn has only a light effect, mainly visible in scenario (b) with an interleaving of departures and arrivals. In this scenario, we observe that the joining nodes obtain identifiers in lower dimensions and, consequently, there are fewer nodes with many aligned neighbors at the end of the experiment. Under scenario (c), the joining nodes have higher chances to populate higher dimensions, thus the percentage of nodes with 13 aligned neighbors remains high.

The joining procedure aims to first populate the higher dimensions, thus, ideally, the $8,192 = 2^{13}$ nodes have identifiers on the highest 13 dimensions. As the identifier space is 2^{31} and because we consider that the dimensions start at 0, the highest 13 dimensions range from 18 to 30. Figure 4.9 depicts the placement of the aligned neighbors, showing the percentage of nodes with an aligned neighbor at entry e. Small entries represent links on small dimensions, pointing towards aligned neighbors that are close in the identifier space, while larger entries represent links on higher dimensions, pointing towards aligned neighbors that are further away. Ideally, the percentages for entries 18 to 30 should be 100%. The figure shows that HyPeer constructs a structure that is almost perfect: with close to 100% of nodes in entries 18 to 30. These percentages remain high (higher than 80%) for the two scenarios under churn. Only few nodes have an aligned neighbor at a low entry, i.e., outside of the highest 13 dimensions. The figure shows thus that our joining algorithm succeeds most of the time in filling the higher dimensions first.

We can thus conclude that HyPeer has a structure that is quite uniform and regular, with a large number of well placed aligned neighbors per node, which is key to deterministically locate redundant paths to be exploited by the different routing strategies.

4.4.2 HyPeer Routing Strategies

We now evaluate the performance of the HyPeer overlay per routing strategy: Gr-HyPeer, FT-HyPeer, LB-HyPeer and Pr-HyPeer. Our baseline is Chord and its greedy routing strategy, with the same identifier space $m = 31$ and $N = 8,192$ nodes, which we refer to as Gr-Chord.

For the analysis of the routing strategies, we define four lookup scenarios, based on two types of structure and two groups of requested keys. The two types of structure are plain HyPeer and its perfect version FH as defined at the end of Section 4.3.1. Besides searching for random keys, we also consider the case where we only look up keys that correspond to the identifier of some node in the network, i.e., we look up nodes. In that case, aligned neighbors are expected to be even more beneficial for lookup because there is no error induced by the difference between the requested key and the destination node: the routing should be more precise and the routing path shorter. The four scenarios are denoted as:

(a) anyID: look up any keys at random;

(b) anyNodeID: look up keys equal to the identifier of some random node in the network;

(c) FH-anyID: like anyID, but in a FH;

(d) FH-anyNodeID: like anyNodeID, but in a FH.

The first lookup scenario is the most general one while the other three cover more specific settings. The last scenario is the ideal case for HyPeer and is thus expected to perform best. For comparison purposes, the results obtained by Gr-Chord in the first two scenarios are reused in the latter two, where the HyPeer routing strategies are run on a FH.

We evaluate the operation of the HyPeer overlay first without considering failures, and then under failure scenarios.

4.4.2.1 Operation without failures

Path Length. Tables 4.3 and 4.4 show a comparison of the path length statistics for all routing strategies, under the four lookup scenarios, when no node failure occurs.

The statistics include the average (Avg), the variance (Var) and the median (Med) of the routing path length.

Routing Strategy	anyID			anyNodeID		
	Avg	Var	Med	Avg	Var	Med
Gr-Chord	6.35	3.00	6.0	6.36	3.02	6.0
Gr-HyPeer	6.54	3.35	7.0	6.61	3.38	7.0
FT-HyPeer	7.85	6.08	8.0	6.86	5.35	7.0
LB-HyPeer	7.37	3.73	7.0	6.67	3.67	7.0
Pr-HyPeer	7.37	3.73	7.0	6.67	3.65	7.0

Table 4.3: Routing path length statistics.

Routing Strategy	FH-anyID			FH-anyNodeID		
	Avg	Var	Med	Avg	Var	Med
Gr-Chord	6.35	3.00	6.0	6.36	3.02	6.0
Gr-HyPeer	6.49	3.23	6.0	6.49	3.24	6.0
FT-HyPeer	7.49	3.28	7.0	6.49	3.24	6.0
LB-HyPeer	7.37	3.42	7.0	6.49	3.24	6.0
Pr-HyPeer	7.37	3.43	7.0	6.49	3.24	6.0

Table 4.4: Routing path length statistics in a FH.

As a general observation, the path length achieved by the HyPeer routing strategies is on average a maximum of 1.5 hop longer than for Gr-Chord. This maximum is observed with FT-HyPeer under the most general lookup scenario, anyID. (Table 4.3). However, when considering Gr-HyPeer, the strategy meant to achieve a short path length, shows roughly the same path length average as Gr-Chord (6.54 compared to 6.35). When searching for node identifiers with the lookup-scenario anyNodeID, the results of HyPeer get closer to those of Gr-Chord, with a maximum increase of 0.5 hop on average. Comparing the results presented in the two tables, we see that the path length is shorter in a FH. In Table 4.4, the results are the same for all routing strategies when issuing requests with FH-anyNodeID. This is due to the fact that all neighbors are aligned and the requests travel exactly towards the destination node.

Figure 4.10 shows the percentage of successful requests per path length using a cumulative distribution function (CDF). The obtained curves are roughly similar, with more differences for larger path lengths. The curve for FT-HyPeer is longer on the right hand side of the graph, where there is a very small number of requests that finds

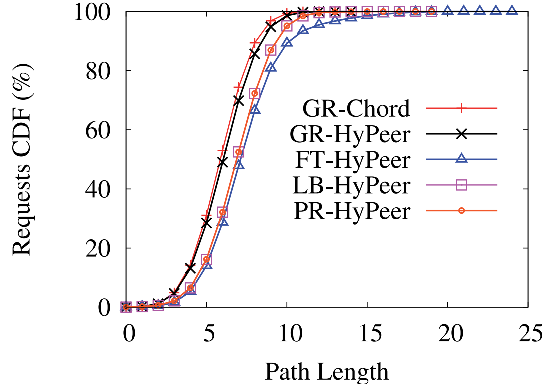


Figure 4.10: CDF of routing path length.

the destinations in 20 to 25 hops. This explains the higher path length average and variance that we have seen in Tables 4.3 and 4.4 for anyID and FH-anyID. The figure also shows GR-HYPEER as the routing strategy that achieves the shortest path length in HYPEER.

Fault Tolerance. In scenarios without failures, there is obviously no need for fault tolerant mechanisms. We note, however, that the path length statistics shown in Tables 4.3 and 4.4 are almost the same for FT-HYPEER as for the other strategies, which makes this routing strategy applicable also in systems with no failures.

Load Balancing. LB-HYPEER has been designed to balance the routing load over the nodes in the system. We can thus expect that, with this routing strategy, the requests (routing load) will be sent through many different (outgoing) links to evenly distribute the traffic. We analyze the average number of links that are used at each node when running the 200,000 requests in Figure 4.11 and the number of forwarded requests that were sent over the most loaded link per node in Figure 4.12.

We have conducted 16 experiments to analyze the average number of links per node followed by the requests. These experiments cover the four lookup scenarios, each under the four load-flows of Table 4.2. The results being similar, we show in Figure 4.11 only the results obtained for LB-HYPEER and GR-Chord for the lookup of random keys (anyID). GR-Chord is affected by the different load-flows, the lowest number of used links being observed under a Zipf-like selection of both the nodes and the keys (zz).

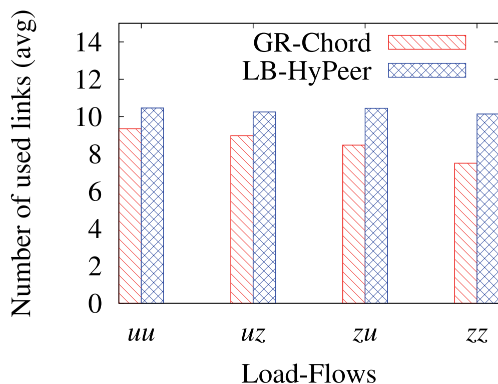


Figure 4.11: Average number of used (outgoing) links per node, under different load-flows.

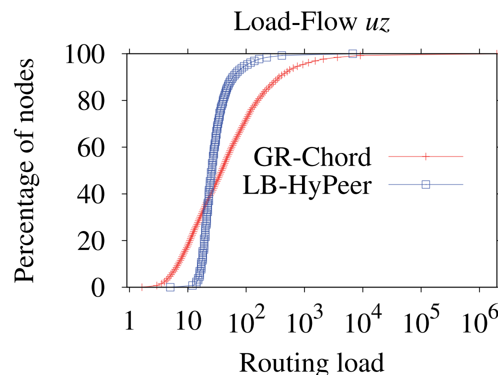


Figure 4.12: Routing load on the most loaded link per node, under popular requests (*uz* load-flow).

This is explained by the path convergence of the requests: all requests for the same popular key tend to follow the same path, and thus the same links are always used. As for LB-HyPeer, the simple technique of random neighbor selection successfully distributes the routing load among many links. Moreover, the average number of links actually used is independent of the load-flow.

We have also analyzed the routing load (number of forwarded requests) on the most loaded link per node under the same settings. LB-HyPeer has consistently shown a low and evenly distributed routing load. Figure 4.12 shows the results after the lookup of random keys in HyPeer under the common *uz* load-flow: a uniform selection of the initiator nodes and a Zipf-like selection of the requested keys. Note the log scale on the horizontal axis. Gr-Chord shows an unbalanced routing load on its most loaded link per node, with extreme low and high values. In contrast, LB-HyPeer balances the routing load, most of the nodes having roughly the same routing load on their most loaded link.

Proximity Routing. Table 4.5 shows a comparison of the average delay time of the requests in HyPeer when different routing strategies are used, under different lookup scenarios. The delay of each request is computed as the sum of the euclidean distance between each two consecutive hops on the request path (each node has random coor-

Routing Strategy	anyID	anyNodeID	FH-anyID	FH-anyNodeID
Gr-Chord	5,560	5,572	5,560	5,572
Gr-HyPeer	5,763	5,815	5,704	5,705
FT-HyPeer	6,910	6,047	6,568	5,705
LB-HyPeer	6,493	5,880	6,472	5,705
Pr-HyPeer	5,089	4,449	4,997	4,215

Table 4.5: Average request delay.

dinates between 0 and 1,000 in a 5-dimensional space). As a general observation, the average path length is smaller when searching for keys corresponding to node identifiers (lookup scenarios anyNodeID and FH-anyNodeID), which explains also the shorter average path delay. Unsurprisingly, for each lookup scenario, the lowest delay time is achieved when the Pr-HyPeer routing strategy is used.

4.4.2.2 Operation with failures

We now analyze the behavior of HyPeer under failure scenarios, with a particular focus on fault tolerance. In particular, we analyze the average path length of successful requests, which is expected to increase given the fact that messages are routed around failures. We show the results of all four lookup scenarios.

The path delay analysis is not included here, since it directly depends on the path length. Likewise, the Pr-HyPeer routing strategy gives results very similar to LB-HyPeer because both produce seemingly random lookup paths (with hops whose length in the logical identifier space do not follow a regular pattern, neither increasing nor decreasing). Therefore, we do not include the results for Pr-HyPeer in our plots.

Figure 4.13 shows the percentage of failed requests when varying the percentage of failed nodes from 0 to 90%. We notice that Gr-Chord yields almost the same results whether looking up random keys or node identifiers. For HyPeer, the first general observation is that all its routing strategies achieve significantly better fault tolerance than Gr-Chord. The best result is obtained by FT-HyPeer, the routing strategy meant to achieve the best fault tolerance. Up to 50% node failures, FT-HyPeer is at least twice as good as Gr-Chord. Under the most general lookup scenario, anyID, only 13% of the requests fail with FT-HyPeer for 50% node failures compared to 33.5% of the requests with Gr-Chord.

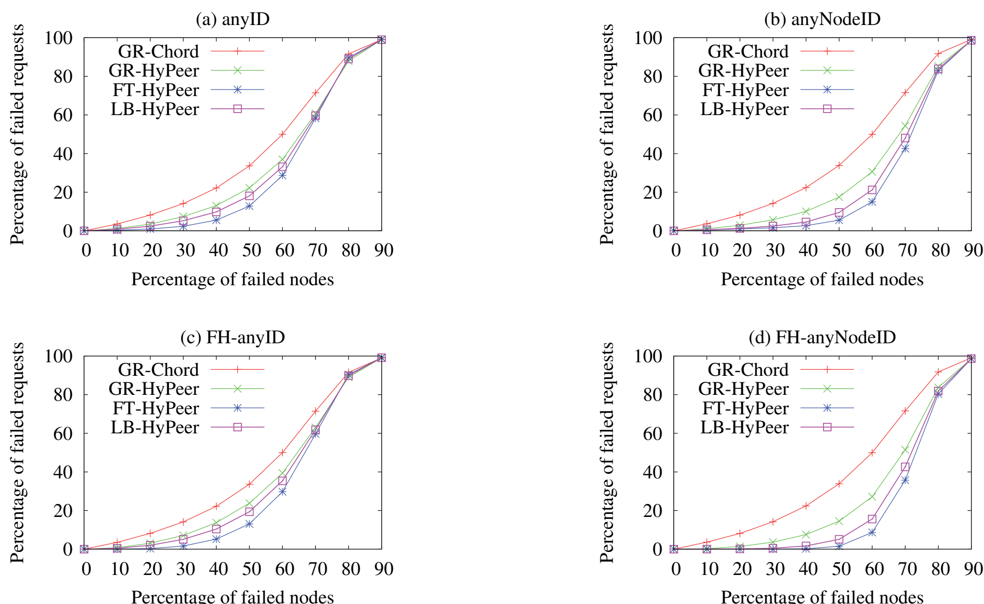


Figure 4.13: Percentages of failed requests under failures.

Moreover, we observe that the degree of fault tolerance for HyPeer depends on the choice of aligned neighbors that are used for forwarding requests. The lowest tolerance to faults is found for Gr-HyPeer, which chooses the furthest suitable aligned neighbor. LB-HyPeer and Pr-HyPeer provide better fault tolerance as they mix short and long hops. The best strategy in terms of fault tolerance is FT-HyPeer, because it chooses the closest suitable aligned neighbor, thus leaving a much larger number of possible paths towards the destination.

When looking up node identifiers instead of keys, the performance of the HyPeer strategies (notably FT-HyPeer) improves because we can better predict the path in advance and we are less likely to need small adjustments close to the destination during the last hop(s). In the ideal lookup scenario FH-anyNodeID, less than 10% of the requests fail with FT-HyPeer up to 60% of failed nodes.

Figure 4.14 shows the average path length of the requests, again while varying the percentage of failed nodes from 0 to 90%. As in the results from the previous figure, Gr-Chord features almost the same results when looking up keys or node identifiers. For HyPeer, we first note that when looking up node identifiers (anyNodeID), the average path length is roughly the same for any routing strategy. This happens because there

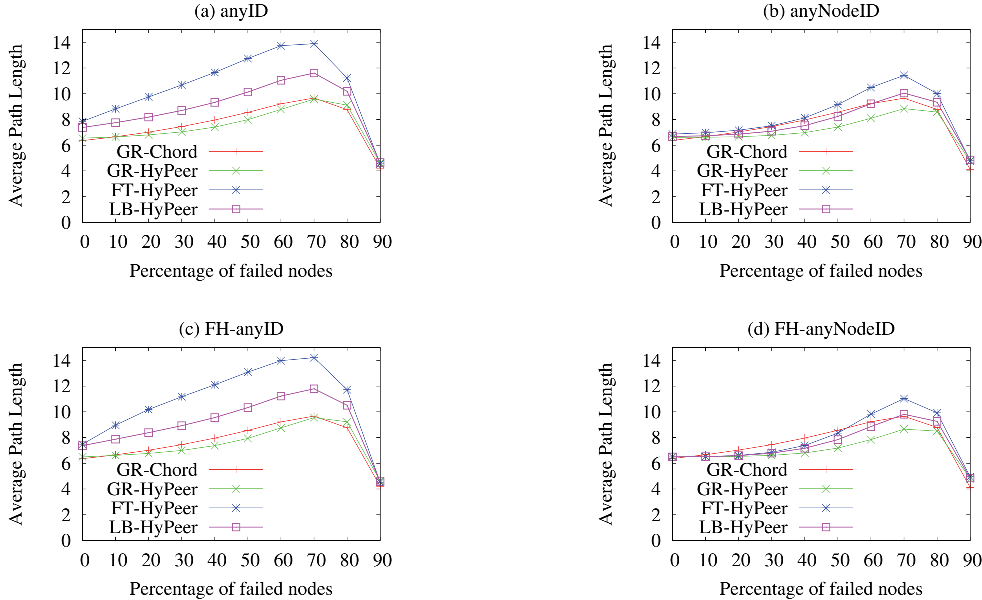


Figure 4.14: Average path length under failures.

is no error induced by the difference between the destination node and the destination key, so the requests sent with the HyPeer routing strategies succeed in almost the same number of hops, even though they follow different paths. It is also interesting to notice here that Gr-HyPeer achieves the shortest path length, even shorter than Gr-Chord, while being more fault tolerant. FT-HyPeer has the highest average path length. This can be explained by the low percentage of failed requests that we have observed in Figure 4.13: requests that succeed with FT-HyPeer but not with other strategies take sometimes very long paths, which increases the average value.

4.4.3 Evaluation Review

With 65%-80% of the nodes having an inter-node distance of 2^{18} and more than 70% of the nodes having 11 to 14 aligned neighbors regardless the churn, we have shown that the HyPeer structure maintains a balanced identifier space and keeps a high number of aligned neighbors. The former property is important for balancing the load of the keys evenly among all nodes. The latter is essential for flexible-choice routing: a high number of aligned neighbors offers many choices for the next hop during routing.

We have presented four routing strategies in HyPeer and shown that each one

achieves its goal. Acting in a greedy manner, GR-HYPEER achieves the shortest path lengths even under churn. Maintaining at each step the highest number of alternative paths in case of failure, FT-HYPEER provides the highest degree of fault tolerance, with only a small increase in path length. Under any load-flow, LB-HYPEER manages to evenly spread the load among the neighbors of the nodes on the routing paths. For proximity routing, the paths traversed by PR-HYPEER have the shortest delays.

We have compared our routing strategies with greedy routing on Chord (that we have referred to as GR-Chord) and we have seen that in terms of path length, GR-HYPEER is similar to GR-Chord, while for the other goals, the HYPEER routing strategies are significantly better. In terms of fault tolerance, with FT-HYPEER there are less than 5% of request failures until 40% node failures, while for 50%-60% node failures, FT-HYPEER is two times better than GR-Chord, while increasing the path length by only 2 to 5 hops on average. LB-HYPEER is better balancing the routing traffic by using up to 3 more links on average for forwarding the routing traffic. Moreover, when taking into consideration the highest load that can occur on the links of the nodes, there are only 45 forwarded requests on average with LB-HYPEER, while for GR-Chord the average is much higher, namely 1,454 forwarded requests during our experiments. Under no failures, PR-HYPEER reduces the delay by 8.5% when looking for any keys on HYPEER, its efficiency going as high as 24.5% when looking up keys with identifiers on nodes on a FH.

4.5 Summary and Discussion

We have proposed a simple peer-to-peer overlay that is able to support different routing strategies, all based on the idea of sending the requests along paths with hops of varying distances, equal to powers of 2, but taken in different orders. Our structure controls the node placement in its identifier space, with neighbors evenly placed at distances also equal to powers of 2. The resulting overlay can thus be thought of as an approximation of a hypercube. Experimental evaluation has shown that the structure is uniform and regular, which is key to deterministically locating alternative paths and route around failures.

We have leveraged the awareness of the nodes placement in order to achieve short average path length, fault tolerance, routing load balancing and low average path delay,

depending on the routing strategy being used. The evaluation of these four HyPeer routing strategies has shown that each one achieves its goal, without increasing substantially the average path length.

For load balancing in particular, we have seen that simply using randomness has already given good results. The variations in load on the outgoing links are significantly reduced. As alternative to randomness, which could improve even more the performances, mechanisms that take into consideration system parameters could be used. Such case would be considering the load on each outgoing link or the load on each node, observed locally or through special messages. This kind of solution would give the advantage of always choosing the least loaded path, however the expense of extra storage or additional messages is not clear if it would be fully justified.

Chapter 5

Adaptive Load Balancing by Link Reorganization

"The key to success is often the ability to adapt."
Anonymous

5.1 Introduction

[1] This chapter presents a study on the load imbalance caused by object popularity in structured peer-to-peer systems and our approach for solving this problem through a novel concept: link reorganization.

Several strategies have been proposed to improve load balancing by adjusting the distribution of the objects and the reorganization of the nodes in the system (see Chapter 3). However, such techniques do not satisfactorily deal with the dynamics of the system, or heavy bias and fluctuations in the popularity distribution. In particular, requests in many P2P systems have been shown to follow a Zipf-like distribution [31], with relatively few highly popular objects being requested most of the times. Consequently, the system shows a heavy lookup traffic load at the nodes responsible for popular objects, as well as at the intermediary nodes on the lookup paths to those nodes.

In order to analyze the object popularity impact on the load distribution we have conducted some experiments of biased request workloads. As expected, simulation

[1] The load balancing solutions exposed in this chapter were presented at the MCETECH [75] and ICCCN [5] conferences, as well as published in the IEEE Internet Computing journal [76].

results demonstrated that, with a random uniform placement of the objects and a Zipf selection of the requested objects, the *request load* on the nodes also follows a Zipf law. More interestingly, the *routing load* resulting from the forwarded messages along multi-hop lookup paths exhibits similar powerlaw characteristics, but with an intensity that decreases with the hop distance from the destination node. Once the destination is reached, the process of downloading files is out of band, therefore not considered in this study.

Based on our analysis, we propose a novel approach for balancing the system load, by taking into account object popularity for routing. We dynamically reorganize the "long range links" in the routing tables to reduce the routing load of the nodes that have a high request load, so as to compensate for the bias in object popularity. We propose thus *link reorganization* (`LR`) in the overlay by a continuous update of the routing tables. In addition, we propose to complement this approach by caching the most popular objects along the lookup routes in order to reduce the request load in the nodes that hold those objects.

Our solution has the following characteristics:

- *minimum impact on the overlay*: no changes to the topology of the system, nor to the association rules (placement) of the objects to the nodes are necessary;
- *low overhead*: no extra messages are added to the system, except for caching. If a node has free storage space, it can dedicate a part of it for caching, which will lead to better load balancing in the system. Other nodes can simply ignore the caching requests;
- *high scalability*: the decision to perform link reorganization or to cache objects are local;
- *high adaptivity*: link reorganization and caching adapt to the popularity dynamics of the objects in the system.

This chapter is organized as follows. In Section 5.2 we introduce the characteristics of the structured peer-to-peer system taken into consideration in this work, then we present simulations showing that a Zipf distribution of requests results in an uneven

request and routing load in the system. In Sections 5.3 and 5.4 we present, respectively, our approach for popularity-based load balancing and its evaluation. Section 5.5 concludes the chapter, with a summary and a discussion on our load balancing solution.

5.2 Preliminaries

5.2.1 System Model

As a DHT model for our study we used Pastry for its particularity of *prefix routing*. A detailed description of the Pastry DHT can be found in Section 2.1.

Basically, in the routing table each entry must contain a node whose ID has a common prefix of a given length (depending on the row number of the entry) with the current node ID and a different value for the following digit. Note that there are typically more than one node satisfying the rule for an entry. In Pastry, the selection of the node for each entry is based on a proximity metric. In our system we propose to reorganize the links by selecting the nodes with the lowest load. Our load balancing solution can be applied to any DHT with neighbor selection flexibility [30], since it is only a matter of neighbor choice.

For the purpose of this study, we assume the system has the following characteristics:

- *stability*: as churn is not expected to affect the load balancing significantly, no node joins nor leaves the system. As a consequence, we do neither consider a retry mechanism upon lookup failure, nor a bootstrap mechanism to join the system. We briefly discuss the implications of churn at the end of Section 5.3.1;
- *homogeneity*: same characteristics for all nodes (CPU, memory, storage size), same bandwidth for all links, and same size for all objects;
- *no locality*: no topology aware routing in the system.

We do not expect these limitations on the system architecture to reduce the effectiveness of our load balancing algorithm.

5.2.2 Implications of Zipf-like requests

Similarly to Web requests [9], the popularity of the objects in many DHTs follows a Zipf-like distribution [31]. This means that the relative probability of a request for

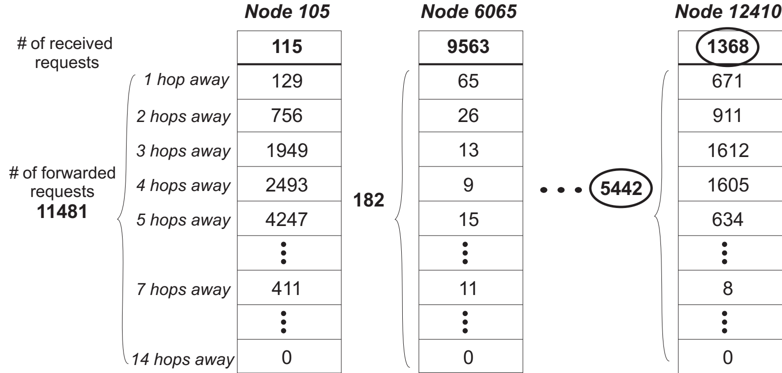


Figure 5.1: Statistics of received and forwarded requests.

the *ith* most popular object is proportional to $1/i^{\alpha}$, where α is a parameter of the distribution, resulting in hot-spots for the nodes that hold the most popular objects.

In case of file sharing applications, many studies have observed that the request distribution has two distinct parts. Very popular files are equally popular, resulting in a linear distribution and less popular files follow a Zipf-like distribution. This usually happens because of the immutability of the objects in file sharing where the clients will request the object only once and then download it (out of band) [29, 50, 84].

In both cases, Web requests and file sharing applications, the amount of traffic received and forwarded by some nodes is much higher than for other nodes. In this context, we analyze the worst case (Zipf-like distribution) and we focus on improving the degraded performance caused by hot-spots.

Each node n_i has a capacity for serving requests c_i, which corresponds to the maximum amount of load that it can support. In our study, we consider the load as the number of received and forwarded requests per unit of time. Some nodes hold more popular objects than others (i.e., have a higher number of received requests), thus being overloaded, with a load $\ell_i \gg c_i$. Other nodes hold less or no popular objects thus presenting a small load compared to their capacity $c_i \gg \ell_i$. With a random uniform placement of the objects and a Zipf-like selection of the requested objects, the *request load* on the nodes also follows a Zipf law. Consequently, we expect that the *routing load* resulting from message forwarding along intermediary nodes to the popular objects also exhibits powerlaw characteristics.

To better understand this problem, we have performed simulations to gather request

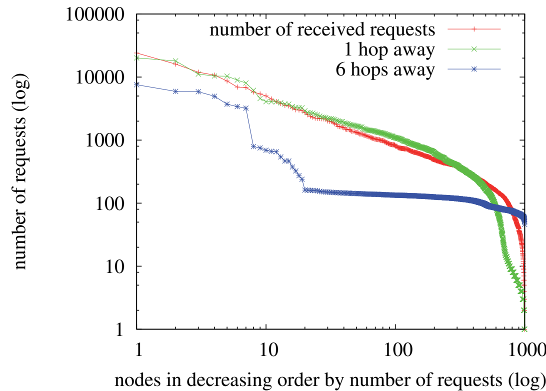


Figure 5.2: Request and routing load.

load information associated to the nodes in the system. At each node, we keep track of the number of requests received for a local object, as well as the number of requests forwarded for a destination located i hops away from the current node. Figure 5.1 illustrates part of the results of a simulation for an overlay network with 1,000 nodes, 20,000 objects randomly and uniformly distributed, and 100,000 requests following a Zipf-like distribution.

The value of an *i-hop away* entry represents the number of requests that it forwards for nodes at a distance of i hops. Node 105 receives only few requests, but it forwards many requests. Conversely, node 6,065 holds a popular object. It thus receives many requests, but forwards only few requests since it is not on a path to a popular object. Node 12,410 presents both a high request load and a high routing load. Obviously, nodes 6,065 and 12,410 become hot-spots.

Figure 5.2 compares the number of requests received by each node, as well as the number of requests it must forward for a destination node that is 1 hop and 6 hops away. All three sources of load follow a Zipf-like distribution, but with an intensity that decreases with the distance from the destination.

5.3 Link Reorganization Solution

In this section we present our load balancing solution that aims to equilibrating the load (routing and request) of all nodes in the system. We introduce the concept of *link*

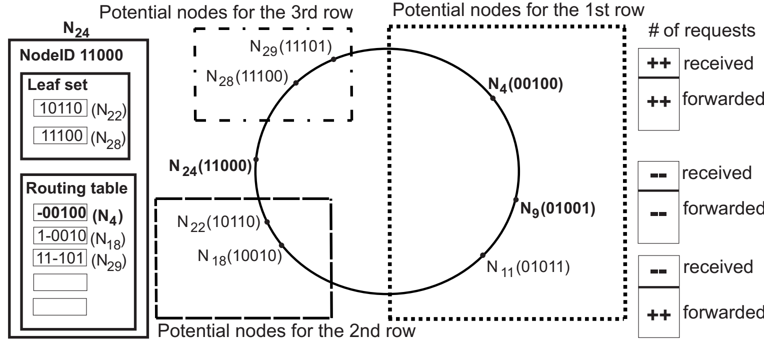


Figure 5.3: Routing tables before reorganization for load balancing.

reorganization, in a load balancing algorithm that we refer to as LR (Link Reorganization). Following to that, we add a caching mechanism as complementary solution in order to deal with extreme high request load.

5.3.1 The LR Algorithm

The key principle of our approach is to dynamically reorganize the "long range links" (i.e., update the routing tables with other long-range neighbors) in order to reduce the routing load of the nodes that have a high request load, so as to compensate for the bias in object popularity.

As previously mentioned, each entry of a routing table can be occupied by any one of a set of nodes that share a common prefix. In our approach, we reorganize the links by choosing the nodes with the lowest (request and routing) load in order to offload the most heavily-loaded nodes. The overloaded nodes (as a consequence of a popular object, or too many forwarded requests or both) are removed from the other nodes' routing tables in order to reduce their load. Instead, the entry will contain another node, from the same region (same prefix), which is less loaded. This way, the nodes that have a high request load will have a small routing load, and the nodes with low request load will share the routing load.

Figure 5.3 shows an example of routing table update with no load balancing and Figure 5.4 illustrates the update based on our load balancing mechanism. In the figures, "++" indicates a high load and "−−" a low one. In the example, node N_4 holds a popular object resulting in a high request load. Since it is a heavily-loaded node, our load balancing solution will remove it from the other nodes routing tables: node N_{24}

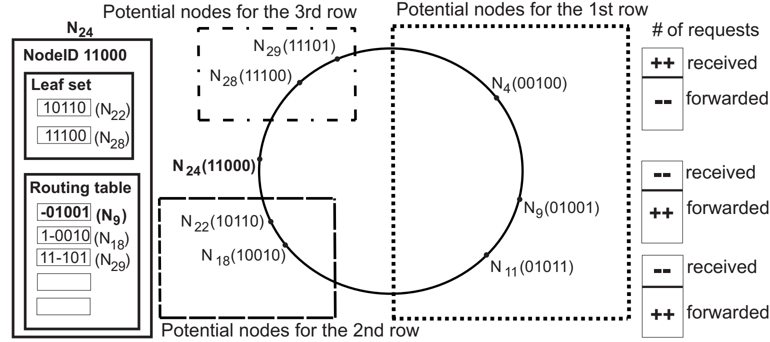


Figure 5.4: Routing tables after reorganization for load balancing.

will update its first entry with node N_9, which is less loaded than node N_4. Consequently, the load of node N_4 will decrease, and the load of node N_9 will increase, thus equilibrating the load in the system.

The routing table updates are performed dynamically while routing the requests, without increasing the number of messages. The `LR` algorithm is shown in pseudo-code in Algorithm 5.1, for a node n_i that forwards a request.

Every node keeps track of the approximate load ℓ_k of each other node n_k in its routing table. Before forwarding (or sending) a request message, each node adds its load to the message (line 17). The ith node in the request path receives the load information of i other nodes in the request message. A node n_i that receives the request, besides handling it, uses the load information in the message to possibly update its routing table. Each node n_j in the message can match exactly one entry in the routing table of node n_i. If the load is lower for node n_j than for node n_k found in the routing table, the entry is updated with n_j (lines 5-8).

The load information corresponding to the entries in the routing table of node n_i is not accurate, since the node cannot know at each moment the real values for the load of each entry. In order to compensate for this limitation, we use several techniques:

- even if the loads for the two nodes n_j and n_k are equal, the entry is updated, since load l_j is n_j's real load but l_k is only an estimation of n_k's load (the condition at line 5 accepts also equality);

- if n_i receives the load information of a node that is already in its routing table (i.e, node n_j is the same as node n_k), its load is updated (line 10);

Algorithm 5.1 Pseudo-code for the `LR` algorithm at node n_i

```
0: {Receive request}
1: for each (n_j, ℓ_j) in the message do
2:    entry ← matching entry for n_j in the routing table
3:    n_k ← current node at entry
4:    if n_j ≠ n_k then
5:       if ℓ_j <= ℓ_k then
6:          Replace n_k by n_j at entry
7:          Store ℓ_j at entry
8:       end if
9:    else
10:      Store ℓ_j at entry
11:   end if
12: end for
13:
14: if n_i not owner of requested object then
15:   n_k ← next node to forward request
16:   ℓ_k ← ℓ_k + e
17:   {Add (n_i, ℓ_i) to the request message to be forwarded}
18: end if
```

- when node n_i forwards (sends) a request to a node n_k, n_i updates the load information for n_k in the routing table using an estimation e of the load of n_k (line 16).

In our experiments, we use an estimation e of 1, since we know exactly that the load of n_k will increment by at least 1 from the request that n_i forwards.

The small amount of data that is added to the message in the form of $O(log_{2^b}N)$ pairs of integers (node and its load) won't typically add extra packets at the network layer and should thus have no effect on bandwidth. With respect to the CPU load, the routing-table update mechanisms add only negligible overhead - in fact, they indirectly decrease CPU load for overloaded nodes by reducing the number of messages to process.

Churn can be easily handled as follows. When a new node n_i joins the system, it populates its routing table in a similar way as Pastry, by collecting routing table entries from the nodes that the joining request goes through. Similarly, the initial load l_i of the new node becomes the average of the loads of those nodes. Both the routing table and l_i are updated over time according to the load algorithm. Node departures do not

require special handling.

5.3.2 Adding Caching as Complementary Solution (LR+C)

The link reorganization solution permits us to balance the forwarding traffic of the nodes in the overlay, but the traffic resulting from the received requests still leads to a bottleneck at the destination node. In this subsection we propose caching as a complementary feature to link reorganization, in order to minimize the number of received requests at the nodes holding popular objects. As a consequence, the request traffic for each cached object will be shared among the node owning the object and the nodes holding the replicas.

Basically, there are two ways to initiate caching: by the client that requests the object and by the server that holds the object [56]. Client-initiated caching is not adequate for applications such as file sharing because a client usually requests an object only once. Therefore, in our approach, the server replicates the object to be cached on some other node(s) in an attempt to reduce its request load. When a request arrives at a node that holds a replica of the requested object in its cache, that node can directly respond to the request.

We refer to two kinds of objects that a node holds:

- *owned object*: an object that belongs to the node according to the initial mapping of objects to nodes;
- *cached object*: a replica of an object owned by another node.

The caching algorithm is shown in pseudo-code in Algorithm 5.2. We make use of two types of counters for the received requests at each node: a per-object counter (for the number of received requests for each object held by the node) and a per-node counter (for the total number of received requests). The counters are incremented at each received request (lines 1-2). A threshold is defined for the per-node counter. Each time the threshold is reached (line 4), a weight is computed for each object held by the node based on the per-object counters (lines 5-12); then, the counters are reset (lines 19-23). The most popular object m_p_o on the node is the object with the highest weight (line 14).

To compute the popularity of an object, we use a weighted moving average, where the weight is computed as a combination of its previous value and the value computed

Algorithm 5.2 Pseudo-code for the caching algorithm at node n_i

0: {Once a caching request from a node n_j is received, increment counters: }
1: $req_recv[requested_object][n_j]{+}{+}$
2: $req_recv_counter{+}{+}$
3: {Proceed only if the threshold is reached: }
4: **if** $req_recv_counter = T$ **then**
5:    {Compute weights: }
6:    **for all** objects o on node n_i **do**
7:       **if** o is the last cached object **then**
8:          $w[o] \leftarrow req_recv[o]/T$
9:       **else**
10:          $w[o] \leftarrow w[o] * \beta + (req_recv[o]/T) * (1 - \beta)$
11:       **end if**
12:    **end for**
13:    {Issue a caching request if needed: }
14:    **if** n_i is loaded **then**
15:       $m_p_o \leftarrow o$, where $w[o]$ is max
16:       $n_c \leftarrow n$, where $req_recv[m_p_o][n]$ is max
17:       send a request to n_c to cache m_p_o
18:    **end if**
19:    {Reset counters: }
20:    **for all** objects o on node n_i **do**
21:       $req_recv[o] \leftarrow 0$
22:    **end for**
23:    $req_recv_counter \leftarrow 0$
24: **end if**

over the last period (line 10). We use a β value of 0.9, such that both terms count in the computation of the object's weight, but the old value (which is a stable information) counts more than the new one. However, after a caching request has been issued for an object, only the new value is considered (line 8).

For the caching mechanism, the following considerations must be taken into account: storage size of the cache and its policies, when to cache an object and where to store it. In the following we detail all these aspects.

The cache and its policies. Each node has a cache (storage capacity) for C replicas. Whenever a caching request is received and the storage capacity is exhausted, the replica entry with the lowest weight is discarded and the new replica is stored.

When to cache an object. A caching request is issued each time the threshold T is reached in case the node is loaded (lines 13-18). Obviously, if the node is not loaded, no caching request is issued, at least until the next threshold.

To know whether a node is loaded or not, we perform two checks:

- *if the node is globally loaded.* We use the load information of the nodes in the routing table; this is not an up-to-date information, yet a rather good estimation of the load of some nodes in the system. A node is globally loaded if its load is bigger than the average load of these nodes;

- *if the node has a lot of received requests.* A node would have a balanced load if the number of received requests is equal to the number of forwarded requests divided by the average path length. Therefore, we compute the average path length of the requests that the node received between two consecutive thresholds. To justify a caching request, a node must satisfy the following condition:

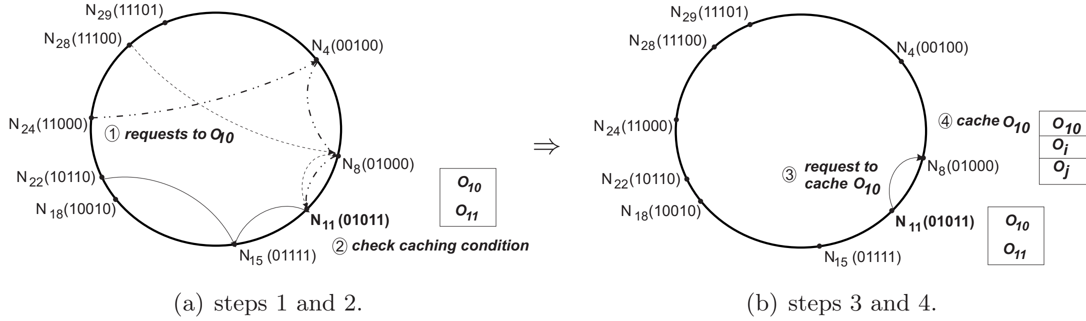
$$recv_requests > fwd_requests/path_avg,$$

  where the counters for the number of received and forwarded requests are reset after each threshold.

If both conditions are satisfied, a node will issue a caching request.

Where to store the replica. Since every message contains information about the request path, the most suitable method is to cache along that path. This can be done (1) on all the nodes in the request path, (2) close to the destination node, (3) at the node that requested the object, or (4) randomly. We choose to do the caching close to the destination node, at the last node in the request path. This has the advantage that the object is cached in the neighborhood of its owner where with prefix routing the possibility for a request to hit a replica is much bigger than elsewhere in the system.

Since the requests for a given object may come from any node in the system, the last hop will not always be the same. The node that is chosen for caching the object is the one that most frequently served as last hop for this object in the lookup paths (line 15).



(a) steps 1 and 2.

(b) steps 3 and 4.

Figure 5.5: Example of functionality of the caching method.

Figure 5.5 presents an example of the caching mechanism. Many nodes send requests for the object O_{10} (step 1), which lies on node N_{11}. After N_{11} receives the requests, it checks the caching condition (step 2), which in this example is true. It computes thus its most popular object, O_{10}, and the node where to store a replica, N_8. Then, it issues a caching request for object O_{10} to node N_8 (step 3). Finally, N_8 caches the object O_{10} (step 4).

We refer to the `LR` algorithm complemented with this caching mechanism as `LR+C`.

5.4 Evaluation

In this section we evaluate our approach by means of simulations. First, we present results of the experiments that we have conducted to analyze our link reorganization solution. Next, we evaluate caching as complementary solution. Finally, some statistics with different Zipf distributions are presented.

The simulated system is an overlay network with 10^3 nodes and $2{\times}10^4$ keys, and we issue $5{\times}10^5$ requests from random sources. Unless otherwise specified, the requested objects follow a Zipf distribution with parameter $\alpha = 1$ and the routing mechanism is based on Pastry with a leaf set size of |L|=4 entries.

5.4.1 Evaluation of `LR`

To analyze the load balancing algorithms, we use the same experimental setup while applying different routing strategies:

- **run0**: as a base for comparison, the experiment is run with no load balancing;

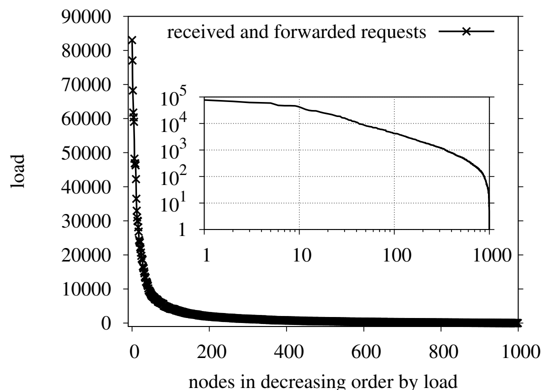


Figure 5.6: Load distribution without load balancing (run0).

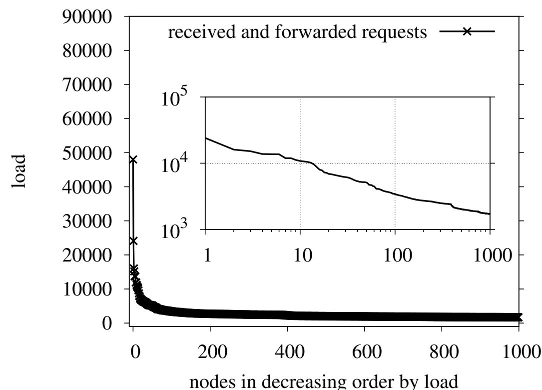


Figure 5.7: Load distribution with load balancing (run2).

- **run1**: dynamic run, where the routing tables are dynamically updated while handling the requests;
- **run2**: same as run1, with the difference that the experiment starts with the routing tables obtained after run1.
- **run3**: static run, with no routing table updates at all, where the experiment starts with the routing tables obtained after run2.

The results are evaluated after each run of the experiment. The first dynamic run (run1) shows the efficiency of the routing table updates. The second dynamic run (run2) simulates the continuation of the first dynamic run, while running the same requests. The purpose of the last run (run3) is to show that in a system with no load balancing strategy, the results are better when starting with optimized routing tables.

In the following experiments of this subsection, the nodes and key IDs are computed on $m = 15$ bits in base $2^b = 2$.

The selection of the keys in the requests follows a Zipf distribution and, as a consequence, the same applies for the load distribution in the system, as can be seen in Figures 5.6 and 5.7, where the nodes are ordered in decreasing order of load. The load represents the number of received and forwarded requests.

Figure 5.6 shows the load distribution with no load balancing (run0). The load is not evenly distributed among the nodes: some of the nodes have very high load (the

left side of the graph), and other nodes have just a small load or no load at all (the right side of the graph).

Figure 5.7 shows the load distribution in exactly the same system, but with our solution taking into account the request popularity, after the second dynamic run of the experiment (run2). As shown in the graph, the highest load is decreased by half. Moreover, the load in Figure 5.6 tends to 0, while in Figure 5.7 it remains almost constant (approximately from node 300), showing that most of the nodes have the same load. The improvement factor is even more visible with logarithmic scales (inset graphs in Figures 5.6 and 5.7).

In order to assure that our solution does not add too much processing by its updates, Figure 5.8 shows the rate of updates to the routing tables in the second dynamic run (run2) for the first 200 time units. The rate of updates is high in the beginning, many nodes updating their routing tables, but it quickly stabilizes at a small value.

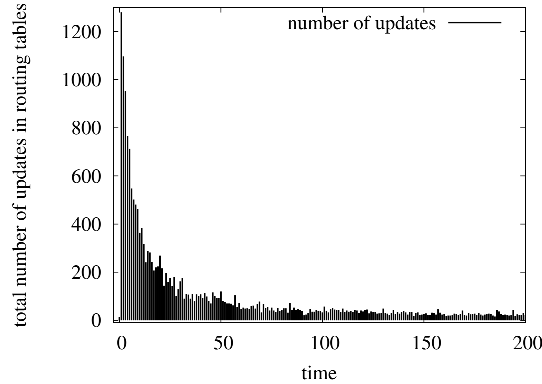


Figure 5.8: Evolution of the number of updates over time (100 requests per unit of time) in the first 200 time units (run2)

Our algorithm for dynamically updating the routing tables of the nodes in the system shifts the load from the most loaded nodes to less loaded nodes, by having the less loaded nodes forward most of the traffic instead. This way, the highly loaded nodes will get rid of the traffic that they had to forward, and become less loaded. The solution does not deal with distributing the keys. This problem has already been well studied and can be addressed by using virtual servers [26]. Our techniques cannot decrease the load below the number of requests addressed to a node. Thus, we still have a Zipf-like distribution, but with much lower intensity.

In order to better perceive the load distribution for the most loaded nodes, Figures 5.9 and 5.10 show the same data as Figures 5.6 and 5.7 only for the first 300 nodes. Additionally, they include the number of received requests per node.

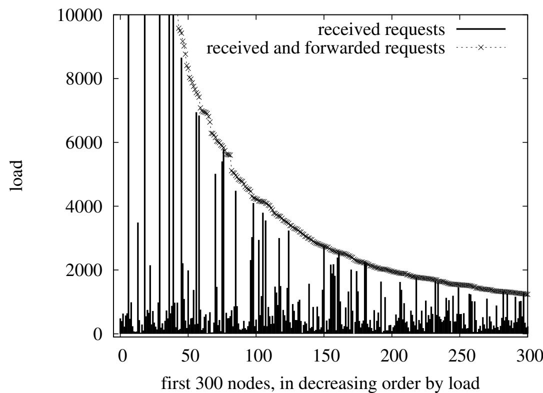


Figure 5.9: The 300 most loaded nodes, without load balancing (run0).

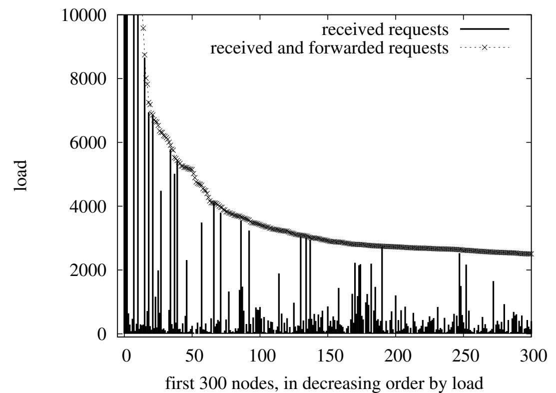


Figure 5.10: The 300 most loaded nodes, with load balancing (run2).

The nodes at the left hand side of the graphs are the most loaded ones. Comparing the two graphs, Figure 5.9 exhibits more nodes with a high load mostly induced by the forwarded requests. In Figure 5.10, fewer nodes have a high load, which mainly results from the received requests. The most loaded nodes are now the nodes with the highest number of received requests; the next most loaded nodes are their direct neighbors. The less loaded nodes at the right hand side of the graph (see Figure 5.9) are now more loaded, which results in a more balanced overall load tending towards a constant (see Figure 5.10).

Until now, we considered a leaf set of 4 nodes. With a larger leaf set of size 8, the results are even better, as the routing load is shared by more nodes in the vicinity of a popular node. These results after two dynamic runs are shown in Figure 5.11.

Table 5.1 contains some statistics (load average and variance) from two experiments. The first experiment (run0) has no load balancing solution. For the second experiment we show the statistics after each of the three runs. Both types of experiments are done for a leaf set of 4 and 8 nodes.

The statistical analysis shows that the variance of the system load is decreasing from 7,161 for the results shown in Figure 5.6 (run0), to 2,167 for the results shown in Figure 5.7 (run2). This confirms that the load extremes are getting closer. The load average is slightly increasing from 2,353 to 2,585, because changing the routing tables in the destination node's closest area might increase in some cases the path length.

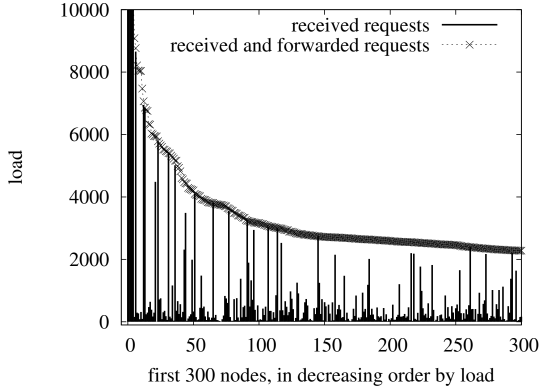


Figure 5.11: Load distribution for the 300 most loaded nodes, using a leaf set of 8 nodes (run2).

Experiment type	Leaf	Avg	Var
run 0: no update	4	2,353	7,161
run 1: update	4	2,535	2,526
run 2: update	4	2,585	2,167
run 3: no update	4	2,648	2,466
run 0: no update	8	2,253	7,103
run 1: update	8	2,319	2,394
run 2: update	8	2,350	1,966
run 3: no update	8	2,383	2,152

Table 5.1: Load Statistics (average and variance) for all runs, using a leaf set of 4 and 8 nodes, respectively.

However, the path length is still in the order of $O\ (log_{2^b}N)$, where N is the number of the different nodes in the system.

The load average is increasing as a consequence of increasing the path length, but the increase is very small. However, we can note a high decrease of the variance. The first dynamic run (run1) has a lower variance than the experiment with no routing table updates (run0). The variance after the second dynamic run (run2), which is using the optimized routing tables obtained from the first dynamic run, is even lower. Comparing the results obtained without updating the routing tables, the variance is much lower in the case optimised routing tables are used (run1 vs. run3).

In the next subsection we present the results for the routing table reorganization strategy complemented by caching, in order to reduce the request load (i.e., the load observed at the left hand side of Figure 5.7).

5.4.2 Evaluation of `LR+C`

The experiments for `LR+C` are done on the same system, this time using identifiers on $m = 16$ bits, with no particular reason.

The results of `LR` are shown in Figures 5.12 and 5.13 after run1 and run2, respectively. The graphs show the load for the first 100 most loaded nodes, while the inner graphs present a global view of the load of all nodes in the system.

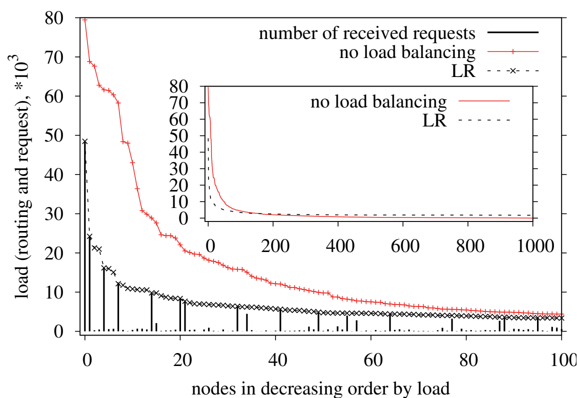


Figure 5.12: Load balancing using dynamic routing table updates (`LR` run1).

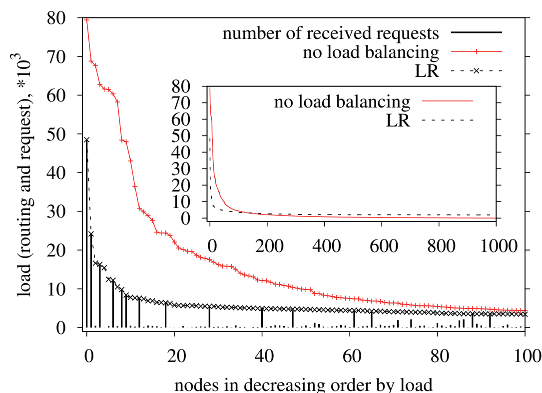


Figure 5.13: Load balancing using dynamic routing table updates (`LR` run2).

The left hand side of the graph in Figure 5.12 shows that the most loaded nodes are the ones with the highest number of received requests (see the vertical lines). Their load cannot be minimized by link reorganization, because the number of received requests cannot be decreased (our `LR` solution applies to the routing load only, not to the request load). The inner graph confirms that most of the nodes have roughly the same load, thus reaching a good level of load balancing. Their number is even higher after run2 (see Figure 5.13).

In our experiments, we used a cache with storage size $C = 3$ and a threshold of $T = 500$ requests. The results using `LR+C` (i.e., both solutions: link reorganization and caching) are shown in Figures 5.14 and 5.15, after run1 and run2, respectively. The experiments were done in the same conditions as before, which allows us to distinguish the benefits of using caching as a complementary solution. Comparing Figure 5.12 with Figure 5.14 after run1 (or Figure 5.13 with Figure 5.15 after run2), we note the improvement in load balancing for the most loaded nodes (left-hand sides of the graphs), where the load dramatically decreases; the load for the nodes at the right-hand side of the inner graph slightly increases with caching, which demonstrates that nodes share the load more evenly.

A potential source of overhead resides in the additional messages sent for caching. For the results presented in Figure 5.15, there are 243 extra messages. This is negligible if we take into consideration the number of requests issued (500,000).

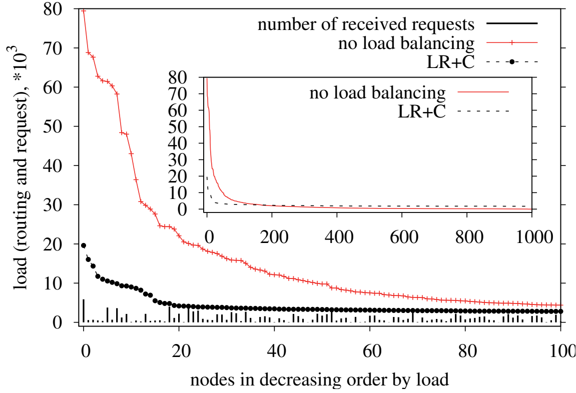


Figure 5.14: Load balancing using dynamic routing table updates and caching (`LR+C` run1).

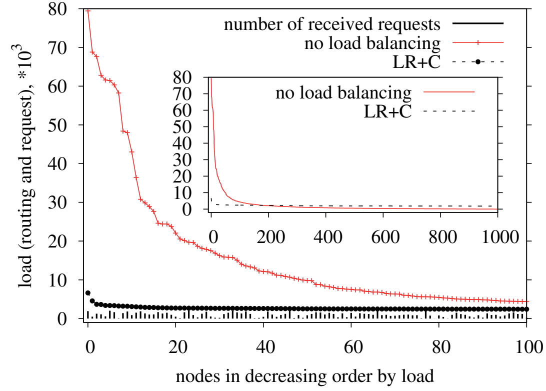


Figure 5.15: Load balancing using dynamic routing table updates and caching (`LR+C` run2).

The two principal variables in the system are the size of the cache C and the value of the threshold T. We have thus conducted several experiments with different values of cache size and threshold to analyze their impact. Table 5.2 presents some statistics of the results obtained while running the experiment with different values for the cache size and the threshold in the same system. The statistics are for the three experiments: run0, run1 and run2. Besides the load average and variance, we also show the number of messages necessary for the caching requests (Msg).

With a cache size of $C = 3$ and a threshold of $Tshd = 500$, we observe that the variance of the system load decreases from 7,243 to 2,056 when using the `LR` strategy (for the results in Figure 5.13). The variance decreases even more to 304 when using `LR+C` (for the results in Figure 5.15). The load average slightly increases as explained earlier in the previous subsection, however with `LR+C` it decreases, because the path becomes shorter in this case.

A smaller value of the threshold means a higher frequency of caching requests, and consequently more messages; however, there is no notable improvement. The cache does not need a large storage capacity to be effective. We obtained the same results when using $C = 3$ and $C = 100$. There is a small improvement when using $C = 3$ over $C = 1$ because the most popular objects can remain permanently in the cache.

For comparison purposes, we also ran an experiment using just caching (`C`), with no

Exp	Cache	Tshd	Msg	Load Avg	Load Var
run0	-	-	-	2,403	7,243
run1					
`LR`	-	-	-	2,505	2,336
`LR+C`	1	500	253	2,248	1,285
`LR+C`	3	500	274	2,214	1,231
`LR+C`	1	1,000	123	2,322	1,373
`LR+C`	3	1,000	123	2,308	1,312
`C`	3	500	639	1,870	6,093
run2					
`LR`	-	-	-	2,563	2,056
`LR+C`	1	500	261	2,099	369
`LR+C`	3	500	243	2,059	304
`LR+C`	1	1,000	151	2,196	465
`LR+C`	3	1,000	134	2,167	380
`C`	3	500	904	1,657	5,871

Table 5.2: Load Statistics.

routing table update strategy. The results show that there is no significant improvement (the variance is still high, 5,871 after run2). This means that caching is no satisfactory solution when used alone.

In these experiments we used a Zipf distribution with parameter $\alpha = 1$ for the request workload. The results using other values for α are shown in the next subsection.

5.4.3 Zipf-like requests with different parameter

The solution that we propose for load balancing is independent of the α parameter of the Zipf distribution. Based on [50] and [9], we performed some experiments varying the value α. The caching storage size is set to $C = 3$ and the threshold to $T = 500$ requests.

Table 5.3 presents the statistics for $\alpha = 0.5$ and $\alpha = 2$. The results are also plotted in Figures 5.16 and 5.17, respectively. For $\alpha = 0.5$, the problem is found in the routing load, which is almost perfectly solved by our routing load balancing solution. The caching complementary solution is not necessary here. As shown in the graph (see Figure 5.16), the results using the `LR` solution and the results using the `LR+C` solution tend to overlap. In the case of $\alpha = 2$, the number of received requests for the most popular objects is very high compared to the other objects; the problem is thus only

	$\alpha = 0.5$			$\alpha = 2.0$		
Exp	**Msg**	**Load Avg**	**Load Var**	**Msg**	**Load Avg**	**Load Var**
run0	-	2,392	6,639	-	2,321	16,568
			run1			
LR	-	2,339	708	-	2,625	13,185
LR+C	110	2,336	684	546	990	2,215
			run2			
LR	-	2,336	96	-	2,683	11,661
LR+C	252	2,318	74	328	719	574

Table 5.3: Load Statistics using $\alpha = 0.5$ and $\alpha = 2.0$ Zipf parameter.

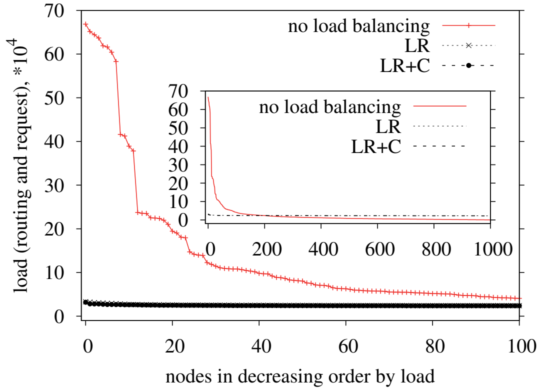


Figure 5.16: LR and LR+C using Zipf's $\alpha = 0.5$ (run2).

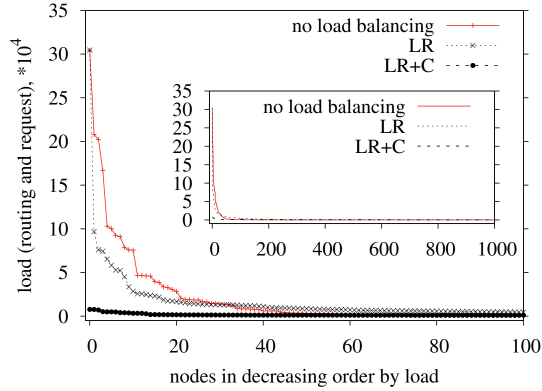


Figure 5.17: LR and LR+C using Zipf's $\alpha = 2$ (run2).

partially solved by the routing load balancing strategy and caching becomes necessary (see Figure 5.17).

5.4.4 Evaluation Review

In our evaluation of the LR and LR+C algorithms, we have performed several experiments under different configurations. We have used four routing strategies (run0 - run3) while using three different α values for the Zipf distribution. The evaluation of the LR algorithm has shown that it significantly reduces the load of the most loaded nodes, while balancing better the load on the rest of the nodes. This happens for the typical value of 1.0 of the α Zipf parameter and especially for lower values, e.g., α=0.5. In such

cases, the variance of the popularity of the objects in the system is low, so the load is mostly given by the routing load. Thus, even with LR only the load can be evenly balanced in the system. For higher variances (e.g., when α=2.0), the load is mostly given by the request load, which makes the LR+C algorithm more adequate for evenly balancing the load between the nodes.

5.5 Summary and Discussion

In this chapter, we have presented an analysis of the load distribution in structured peer-to-peer systems taking into consideration the load caused by the popularity of the objects. Our analysis showed that besides the high request load on some peers, the traffic routing generates a routing load that is not well balanced in the system. Based on these facts, we proposed a novel approach to minimize the load generated by popular requests by reorganizing the routing tables accordingly.

We thus deal at peer side with both *routing load* and *request load*. Since both these loads charge roughly the same a peer (forwarding a request vs. receiving a request and launching the corresponding transfer), we have considered their aggregate when balancing the load. This has the advantage that a peer that is already loaded with request load is freed from routing traffic, our LR solution decreasing the routing load for this peer. It is straight-forward that the load on a peer cannot be reduced less than its request load, because LR only deals with deviating the routing traffic.

The traffic is sent over paths that contain nodes that are less loaded. The algorithm requires neither changes to the topology, nor to the association rules (placement) of the objects to the nodes. It does neither add extra messages, nor significant complexity to the system. The routing table updates are based on local load estimations, as well as the information received with each request, which contains load information about each node in the request path. It is trivial to see that more information about the system is available, the better the lookup traffic can be balanced. Thus, we might also consider adding load information to the response message of the lookup for an object, which could also propagate it further.

As future work, we might consider a combined metric: proximity (as proposed in Pastry) and load information. This could be a trade-off between proximity-aware

routing, reducing the traffic and thus also the load at underlay level, and load-aware routing, balancing the load at overlay level and, to some extent, also at underlay level.

In order to overpass the limitation of LR which cannot reduce the load of a node under its request load, additionally we have proposed adaptive caching based on the popularity of the objects. The caching mechanism uses a weighted moving average of the request load of a peer during the last and the current time period, that avoids high oscillations. The particularity of our caching algorithm is that it makes also use of the load information that travels in the system, to decide locally whether a node is globally loaded, since only the most loaded nodes in the system should issue caching requests. Caching does require some extra messages to be exchanged (the caching requests).

The results from experimental evaluation of LR and its extension LR+C demonstrate a more balanced traffic and, consequently, improved scalability and performance.

Chapter 6

Network-Friendly Gossiping

"You do not know it but you are the talk of all the town."

Publius Ovidius Naso

6.1 Introduction

[1] In this chapter we tackle information dissemination and its impact on the underlying network. During information dissemination, one or several messages containing a specific information need to be transmitted from an initiator node to all the other nodes in the system. This is a frequent task for example in news services, where news need to be reached by all nodes, and moreover quickly and without significant delays between them. Having all nodes involved, the dissemination procedure requires lots of communication, hence good algorithms need to be employed for fast dissemination and preferably through a small number of messages in order not to overload the overlay. Nevertheless, the underlying network can still be burdened: without knowledge about the underlying topology, the packets carrying the information messages might follow long paths in the underlay during the overlay communication paths. Addressing this problem, we come forth with proximity-aware gossip-based protocols that reduce the load at the underlay level, thus calling them *network-friendly*. Our protocols do not affect the dissemination efficiency.

Given the prevalence of P2P traffic in today's Internet, it is important to understand the impact of protocols that are not aware of the underlying network, and develop

[1] The load reduction solution exposed in this chapter was presented at the SSS [80] conference.

solutions to minimize their traffic. This impact can be measured in two main ways: what is the stress imposed on each component of the infrastructure (routers, links), and how is this stress distributed over all components. Such observations give a good idea on the traffic volume and its flow in the underlying network.

Proximity-awareness is mostly available to overlays through proximity metrics, nodes becoming aware of the distance (at the underlay level) to other nodes from the overlay. In an attempt to build efficient peering relations that are proximity-aware, some P2P protocols use application-level measurements, such as the round-trip time (RTT) obtained by ICMP measurements, e.g., as performed during the construction of routing tables in the Pastry distributed hashtable [71]. These techniques are used mostly to enhance the performance as experienced by the user, by reducing communication delays, but it remains unclear whether this approach is really effective at reducing the burden on the network. That is, preferring low-delays routes does not necessarily lead to using shortest paths and it tends to saturate low-delay links and associated routers. Therefore, the length of the communication paths, i.e., the number of traversed routers, is a better indicator of the stress on the infrastructure (a long path obviously loads the network more than a short one). This information is not, however, readily available to the application.

The gossip-based dissemination protocols are very simple yet extremely robust, they are highly dynamic by nature and rely on random interactions with a set of neighbor peers that changes over time, and they are perfectly adapted to large-scale decentralized systems. Most importantly, classical gossip-based protocols are not particularly friendly with the infrastructure (notably because they select random peers to communicate with). In order to make gossip-based dissemination more usable, we propose network-friendly gossip protocols that can use various metrics for selecting, in a semi-random manner, application-level communication links between peers. The objective is to reduce the impact of the dissemination on the infrastructure while keeping good performance. These protocols are based on network-aware peer sampling services and use a combination of push- and pull-based gossiping. We specifically consider metrics based on delay and path length (giving the previous observation on these metrics being different not only in efficiency but also in obtaining measurements) on various topology models, both synthetic and real-world. We study the efficiency of the protocols in terms of performance of dissemination and load volume. As such types of protocols are

lightweight in terms of bandwidth consumption (their primary usage is the robust dissemination of small messages or meta-information with sizes in the order of kilobytes), we do not consider the bandwidth cap of links nor the usage of available bandwidth as being potential sources of bottlenecks in the system.

Related work. A first approach to network awareness consists in introducing a bias in the selection of peers in a peer-to-peer system by leveraging ISPs' knowledge of their infrastructure [1]. The ISP proposes an oracle node which, given a list of IPs, returns this list sorted in decreasing order of network friendliness (as determined by the ISP). While this solution is appealing because it allows more knowledge and control by the infrastructure, in particular regarding the peering relations of the ASes, it is unclear whether ISPs are willing to deploy such services in the near future.

Another approach uses only information from the system itself, where the solutions take place at the application-level. Our solution is also one of them. In [3], synthetic coordinates (mentioned earlier in Chapter 3 and detailed in the following Section) model the delay and the load associated with traffic in a content delivery network. Although this approach allows the network to balance the load on the peers and on the routers, it is only helpful for long-lasting communication patterns with non-fluctuating bandwidth and its applicability to gossip-based dissemination is unclear.

Our contribution focuses on protocols where no stable communication patterns between peers can be leveraged, yet the load has to be reduced and balanced on the infrastructure. Here, bandwidth is not the primary concern, but the presence of a multitude of lightweight operations that, summed up, can represent a considerable load for the network. Knowledge about the network layer structure can also be used to perform various application-level optimization [74] (e.g., balancing the routing load by mapping the routes in the overlay onto those in the infrastructure, or recovering faster from failures). Such knowledge is, nonetheless, not readily available to the application nor easily exploitable in a decentralized manner.

Roadmap. We have already presented the gossip-based protocols, basic gossip-based dissemination and membership management protocols back in Chapter 2. Having the context already defined, we now elaborate on the tools required for network friendliness and present variants of gossip-based dissemination in Section 6.2, from complete

unawareness of the underlying infrastructure to network-friendly solutions. We evaluate the resulting protocols in Section 6.3, in terms of performance and impact on the network using various topologies. Finally, we conclude the chapter in Section 6.4.

6.2 Network Friendliness

For our network-friendly gossip-based dissemination approach, we start on defining "network friendliness" and our metrics and the methodologies that we use for obtaining measurements, then we specify their usage within the push and pull models.

6.2.1 Network-Friendliness through Underlay Proximity Awareness

We define *network friendliness* as the ability of a protocol to limit and if possible to balance the load it imposes on the elements of the network upon which it operates. In this section, we elaborate on the facilities, available to the protocol designer, that we will use for making gossip-based dissemination network friendly.

6.2.1.1 Metrics for Proximity Awareness

We consider a network composed of multiple entities (autonomous systems or ASes, each composed of multiple routers), in which a communication between two nodes follows a path of routers: $n_a \rightarrow r_a \rightarrow ...r_i... \rightarrow r_b \rightarrow n_b$, where $r_a, r_i, ..., r_b$ belong to the set of all routers R. A message of size m between n_a and n_b loads each router on the path by m. Network friendliness aims at (1) reducing the overall load on all routers, i.e., $\sum_{r \in R} load(r)$ and (2) balancing the load on each router, reducing the differences between $load(r)$ for all $r \in R$. Intuitively, one can approach both objectives by preferring short routes and using routers that lie in the vicinity of n_a and n_b (e.g., in the same AS). It is clear that randomly selecting communication partners will lead to configurations that are far from network-friendly, with actual message paths that unnecessarily traverses routers over several continents. We now present the two main metrics that are available to an application for choosing nearby communication partners: using the *delay* as an estimation of the route size, or discovering the actual *route size* by lightweight probing of the IP network. We will use both these metrics in our network-friendly dissemination solution presented in Section 6.2.2.

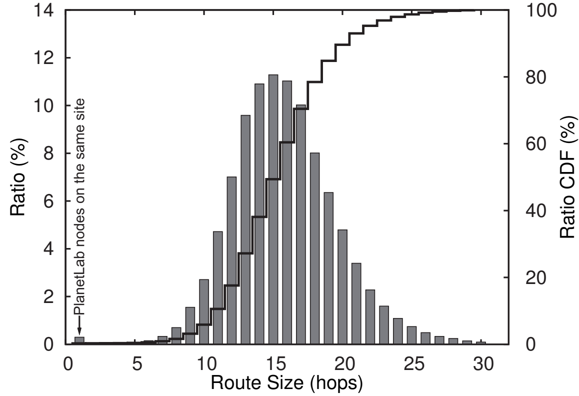
Application level round-trip time (RTT). The first metric, which is also the most commonly used (e.g., when constructing routing tables in Pastry [71]), is the time required for a message to go from a node n_a to another node n_b, estimated as half the round trip time of a small packet exchanged between n_a and n_b (RTT). The RTT can be measured either at the application level, which can benefit from pre-established connections, or by using the ICMP layer (`ping`). Note that this metric is usually leveraged for enhancing the application performance, not for improving network friendliness. While the relation between low delays and low path length may seem intuitive, it is interesting to investigate whether using the RTT estimation for path lengths achieves *the best possible* network friendliness.

Routes lengths. The knowledge of route lengths is usually a metric that is available to the infrastructure manager only, i.e., the Internet service provider (ISP). In the general case, when collaboration between the application and the ISP [1] is not available, it is necessary to *probe* the network for retrieving this information. The *traditional* way of discovering a route (and hence determine its length) is to use the `traceroute` tool. This poses some problems: (1) this tool requires administrator rights, which is not desirable for an application, (2) its load is quite high as it obtains additional information (e.g., the name and addresses of all routers in the path) that are not needed in our context and (3) some networks might block the traceroute requests for security reasons, in order not to reveal their network architecture.

We use instead a lightweight mechanism to obtain route length at the application level, using regular ICMP packets. Every packet sent from a node n_a to another node n_b contains a TTL field. On each router $r_a, r_i, ..., r_b$, IP specifies [8] that the TTL is decreased by the number of seconds the message has passed onto the router, or by one if this time is less than a second. Obviously, the latter case largely dominates in today's Internet, that is, it is safe to consider that on each router the TTL is reduced by 1. Messages that reach TTL 0 are dropped. It follows that a probe message sent with TTL x will only reach its destination if $|r_a, r_i, ..., r_b| \leq x$. The lowest necessary TTL t for successfully *probing* n_b from n_a thus indicates the path length from n_a to n_b. Algorithm 6.1 presents a simple method to determine t. This algorithm uses a divide-and-conquer approach to determine the lowest TTL for a message to reach its destination. The expected median value of the TTL is T which, properly set, allows

Algorithm 6.1 Lightweight probing of path length using the `TTL` field in IP messages (left). Route sizes distribution for 610 PlanetLab [63] nodes (right).

1: $t \leftarrow T$ (expected median length)
2: $r_{\max} \leftarrow 2t$, $w_{\text{found}} \leftarrow \perp$
3: **while** $(r_{\max} - r_{\min}) > 1$ **do**
3: $\{$send a packet with TTL $t\}$
4: **if** no reply within timeout **then**
5: **if** $w_{\text{found}} = \perp$ **then**
6: $r_{\max} \leftarrow \min(r_{\max} + T, 255)$
7: **end if**
8: $r_{\min} \leftarrow t + 1$
9: **else**
10: $r_{\max} \leftarrow t$, $w_{\text{found}} \leftarrow \top$
11: **end if**
12: $t \leftarrow \lfloor \frac{r_{\min}+r_{\max}}{2} \rfloor$
13: **end while**
14: **return** t



for faster probing but this is not a requisite. The expected number of sent probes is $O(\log_2 T)$. We tested this algorithm by discovering the path lengths between all nodes-pair in a set of 610 worldwide nodes on PlanetLab [63]. The method successfully detected all route lengths, in most cases within 5 message exchanges (T was set to 15).

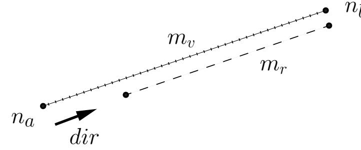
The use of the `TTL` field in TCP packets for discovering infrastructure characteristics without relying on tools such as `traceroute` has been successfully used in a different way by the *Recursive Packet Train* (RPT) method [33], whose goal is to discover bottlenecks (i.e., links that are limiting the overall bandwidth offered by the route) in the path between any two Internet end hosts.

6.2.1.2 Reducing measurement costs using estimations

An overlay substrate for gossip-based dissemination is likely (and willingly) dynamic. It is contrary to the objective of network friendliness, and particularly to the reduction of the load, to have each single peer probe any possible neighbor node it encounters. It is not necessary either to use *perfect* measurements, especially if the act of performing them produces as much load as they were meant to avoid or if the result of the measurement is not constant.

Algorithm 6.2 Vivaldi network coordinates update.

1: {**Local computations:**}
2: $dir \leftarrow u(c_b - c_a)$
3: $m_v \leftarrow ||c_a, c_b||$
4: $diff \leftarrow m_v - m_r$
5: $\delta \leftarrow \max(\delta_{\min}, \delta - \delta_{\text{decr}})$
6: {**Apply force:**}
7: $c_a[i] \leftarrow c_a[i] + dir[i] \times diff \times \delta$



dir: direction unit vector
m_v: Vivaldi estimation
$diff$: estimation vs. measure
δ: adaptive delta
c_a: local coordinates

Network coordinates. An appealing technique for reducing measurements is to use *network coordinates* [13]. The idea is to embed all nodes in a metric space of moderate dimensionality, such that the distance between the points representing two nodes in this space provides a good estimate of the metric (delay or route length). In our evaluations, we use 5 dimensions, as it represents a good tradeoff between the accuracy of the estimations, and associated computational costs and convergence times [13]. Each node "bootstraps" its coordinates by evaluating the metric with a set of landmark nodes. The evolution of coordinates is similar to that of a spring-mass relaxation system: the goal is to reduce, gradually, the differences between the predicted and experienced metrics. Algorithm 6.2 details the coordinate update of a node n_a, after probing the peer n_b chosen by `selectPartner()` for the exchange (in case of delays, this probing is done using application messages, while for path length an explicit probing is needed). Node n_a knows its own coordinate, c_a, and the coordinate of n_b, c_b. The actual measure is m_r while the estimate m_v is given by $||c_a, c_b||$ (distance between n_a's and n_b's coordinates). The difference between m_r and m_v is compensated by slightly moving n_a's coordinate, covering part of the difference between the estimate and the measurement. Note that the whole distance between c_a and c_b is not compensated, in order to avoid too big oscillations of coordinates. Moreover, moving too much the coordinates of n_a in an update would change too much its estimations towards its other neighbors. The portion of the difference is given by δ, which is moderate for the first adjustments (we use $\delta = 0.1$) and decreases down to its minimum value as the node converges towards its "ideal coordinate" for each exchange (we use $\delta_{\text{decr}} = 0.005$ and $\delta_{\min} = 0.05$). The initial setup of the coordinates is done by probing 20 landmarks.

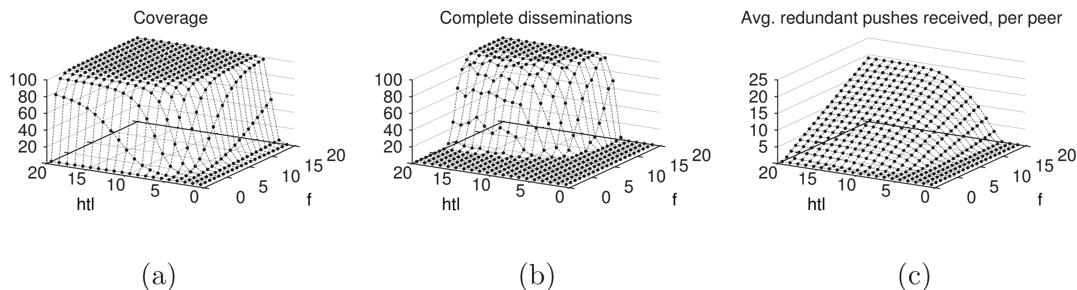
6.2.2 Network-Friendly Gossip-based Dissemination

As previously discussed in Chapter 2, gossip-based dissemination uses two models: push (to send or forward an epidemic message) and pull (to request a dissemination message). Giving the random nature of these models, they often generate useless messages, as sending a epidemic message to a node that already has it (with push, called duplicate message) or sending a request to obtain a epidemic message to a node that does not have it (with pull). Comparing the effects of the useless messages of the two models, a push message generates more load on the network than a pull message, since the former is expected to be larger (i.e., the epidemic message can be large in size), while the latter is smaller, containing only a request.

We now proceed on describing the design of network-friendly gossip-based dissemination protocols. We first discuss the *limited push* approach, intended to reduce the load of duplicate messages, and we elaborate on a network-friendly usage of the two models. Then, we describe the creation of the support overlay, and finally elaborate on the various dissemination scenarios.

Limited push. Gossip-based push propagation reaches all N nodes in the network w.h.p. in $O(\log N)$ rounds if the fanout is $O(\log N)$. During dissemination the number of messages sent by push grows exponentially, and so does the number of *duplicates*, because the probability of reaching an already infected node increases.

Figure 6.1 presents the behavior (simulated, and averaged over 1,000 runs on a 100,000 nodes network) of a push-only dissemination, in terms of coverage and redundancy. The three graphs depict (a) the coverage, as the percentage of nodes that receive the epidemic message, (b) the complete disseminations, as the percentage of runs where the epidemic message was received by all nodes, and (c) the average redundant pushes received per peer. All experiments were done with different values of f and htl, between 0 and 20, in order to observe their influence on the efficiency of the dissemination. In the left-hand side graph we observe that close to 100% coverage is achieved already with $f = 4$ and $htl = 8$, but much higher values of f and htl are required to ensure that 100% of peers get all messages. This can be better seen in the central graph, which shows that a high percentage of complete disseminations are obtained only for the highest values of f and htl. Even though these results seem to be prefer high values, the drawback of their usage is a high number of duplicates, which



(a) (b) (c)

Figure 6.1: Push-only dissemination: coverage (ratio of peers notified), complete (ratio of dissemination that notify *all* peers) and redundancy ratio for various f and htl.

also increases with f and htl, as we can see in the right-hand side graph. This graph shows that the number of duplicates is extremely high even with $f = 4$ and $htl = 8$.

This way of obtaining complete dissemination is contrary to our objective of network friendliness, since we do not want to generate duplicate messages that charge needlessly the network. It means that we cannot use only the push model, and moreover, the fanout and the hops-to-live values have to be carefully selected.

We thus propose to combine push and pull in the following way. An initial, limited push *seeds* the network, leveraging the initial exponential growth phase, but stops before yielding duplicates. Periodic, lightweight pull messages are then used to disseminate the message to all peers. The values of f and htl have to be set properly to reach a certain proportion (e.g., 10%) of the network. Obviously, these values depend on the size of the network, an information that is not known to the peers directly, but that can be calculated by a simple mechanism [52] on top of membership management messages. As the desired coverage is small enough for not having duplicates, the number of nodes touched by a push can be estimated as $\sum_{h=0}^{htl} f^{htl}$. Since f is fixed, a node n_i issuing a new message uses the htl value that best approximates the desired coverage (almost 10% in our experiments). Alternatively, different push messages can be initiated by n_i, with different htl values for getting closer to the target coverage.

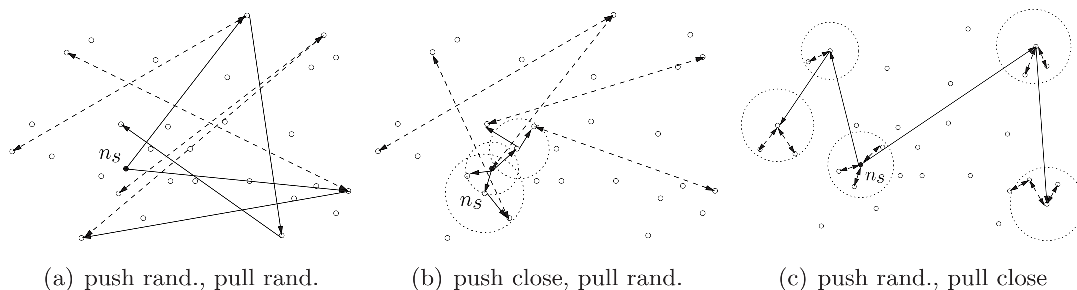
Construction of the support overlay. Gossip-based dissemination relies on some randomness in the peer samples available at each node. We call the set of peers constructed by regular sampling “random peers”, gathered in the “random view”. These

views are constructed using Cyclon [87]. Moreover, we use the T-Man protocol [36] along with Vivaldi coordinates [13] to construct a set of *close* peers, called the "close view" at each node. Both Cyclon and T-Man use views of size 20, and 8 peers are exchanged at each cycle. The proximity function between a node n_a and a potential neighbor n_i is the distance between their coordinates (based on delay or route length). Obviously, the overlay composed of only close peers is not likely to be connected (e.g., all peers from the same institution will form a separate overlay), thus both views are necessary to ensure dissemination termination and we need to use appropriate peer selection strategies, as discussed next.

Peer selection strategies. Classical gossip-based dissemination [7, 19, 44] uses random partner selection for both push and pull. Instead, we choose to use two `selectPartner()` operations, depending on whether the transmission of information is by push (i.e., finding a partner to send data to) or by pull (i.e., finding a partner to ask data from). Each selection can be made in any of the two views (of random or close neighbors), yielding thus four possible strategies. An important point is that a push message is of a greater size than a pull request: the former contains the epidemic message, while the latter only contains some information about the epidemic messages that it has, which for example can be represented using a Bloom filter. We only consider their size, and not the nature of the epidemic message and neither the content of the pull request.

The *push-close/pull-close* strategy can be dismissed right away: the clustering made by the selection of close peers will most likely produce non-connected overlays which, lacking random links, cannot preserve robustness nor ensure termination. The other peer selection strategies are depicted in Figure 6.2. A node n_s is issuing an epidemic message, which is disseminated in the system by push messages (the solid lines) and pull requests (the dashed lines). These strategies differ in the distance that the push and pull messages cover at the underlay level, which can be visualised in the length of the corresponding lines. The vicinity of a node (the underlay "area" which contains its close neighbors) is depicted by a dotted circle.

The *push-random/pull-random (RR)* strategy, depicted in Figure 6.2(a), refers to "classical" gossip without close view. It imposes the highest load on the network, as arbitrarily long routes are used. Nonetheless, its performance is expected to be good as



(a) push rand., pull rand. (b) push close, pull rand. (c) push rand., pull close

Figure 6.2: Peer selection strategies. Dotted circles represent close views. Solid lines correspond to pushes and dashed lines to pulls.

the network is uniformly seeded by the limited push, and the probability to touch an infected node by pull requests does not depend on the position of the requester in the network. This strategy is the baseline for comparing the two network-friendly strategies: *push-close/pull-random (CR)* and *push-random/pull-close (RC)*, shown respectively in Figure 6.2(b) and Figure 6.2(c). CR seeds by limited push the vicinity of the initiator n_s, which has the disadvantage that it increases the risks of duplicates in n_s's vicinity, but presents the advantage of using short paths for the push phase. Intuitively, the better strategy is RC: n_s pays the price of seeding remote nodes through long routes, and these nodes are then in charge of propagating the message by pull requests to their vicinity, using smaller routes. However, if no node has been touched by push in a cluster of nearby nodes (possibly not connected with the rest of the networks by their close views), RC may produce large delays for message delivery, or even lead to some messages not being delivered at all. To avoid this scenario, we force RC to sometimes select a random partner for a pull request (with 5% probability).

6.3 Evaluation

We evaluate metrics and strategies for network-friendly gossiping by the mean of simulations of both the application layer *and* the network layer. Our discrete time simulator considers the network as a set of routers and links between them. Each application node is attached to a router, and messages follow the shortest path in the network in terms of the sums of delays for all traversed links. We chose to use simulation rather than a deployment for being able to compare inter-router delays and routes lengths w.r.t.

the ones estimated by Vivaldi at the application layer. Each application node runs the Vivaldi coordinates system, a Cyclon peer sampling service, our route length measurement algorithm, a T-Man protocol for clustering nodes in local views according to the chosen metric, and the dissemination protocol. To allow for a fair modeling of Vivaldi and the associated estimation error, delays on each link are subject to a ±10% variance.

6.3.1 Topologies

We use both *synthetic* topologies representing classical network models found in the literature, as well as a *real* topology model collected from a university network. All topologies are composed of 651 routers to match the size of the real one.

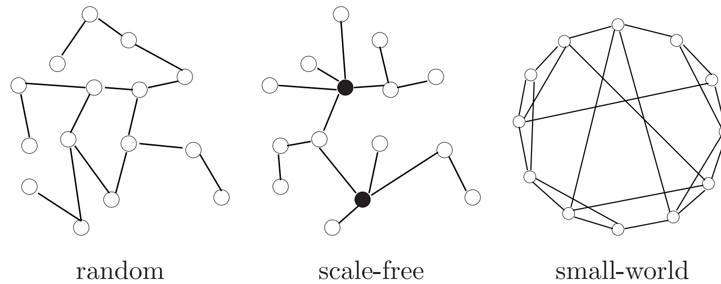
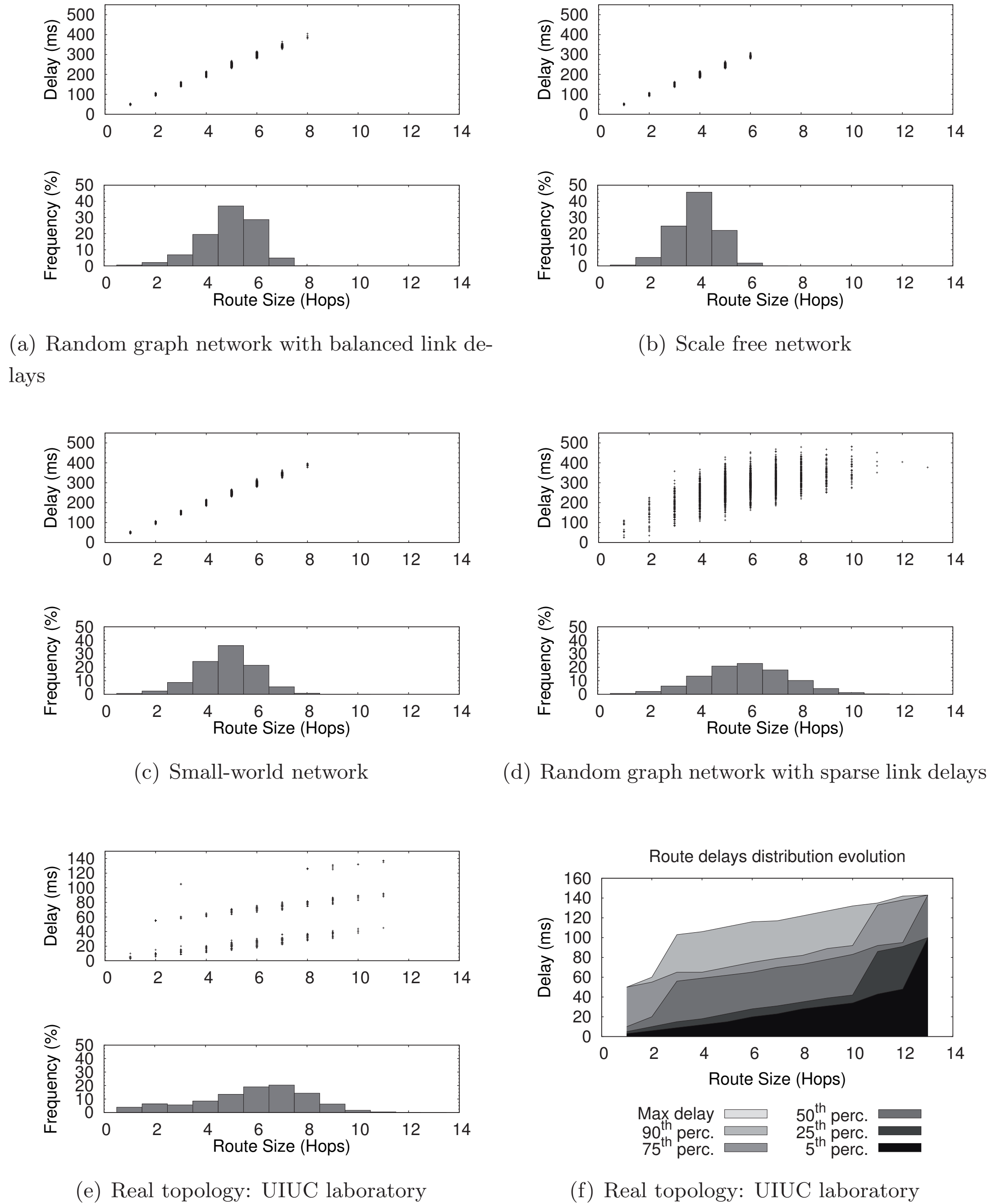


Figure 6.3: Synthetic topologies.

Synthetic topologies are helpful for understanding the behavior of network friendly gossiping in various well-studied graphs models. We use 4 different synthetic topologies that are representative of global characteristics of real networks, and that are the most commonly used for modeling these networks characteristics. They are based on random, scale-free and small-world graphs, as visualized in Figure 6.3. The "random" graph is the only one that does not have any constraints. The "scale-free" graph is characterized by a degree distribution that follows a power law; the nodes with the highest degrees are shown in black in the figure. In the "small-world" graph, most nodes are not neighbors of one another, but most nodes can be reached from every other node through a small number of hops.

All our synthetic topologies are symmetric: when building a link from a router r_a towards a router r_b according to the characteristics of the current topology, a link between r_b and r_a is also additionally created. Unless explicitly noted, all links between routers have a delay of 50 ms ±10%.



(a) Random graph network with balanced link delays

(b) Scale free network

(c) Small-world network

(d) Random graph network with sparse link delays

(e) Real topology: UIUC laboratory

(f) Real topology: UIUC laboratory

Figure 6.4: Characteristics of the five topologies.

Figure 6.4 shows our five topologies (four synthetic and one real), together with their characteristics. For each type of network we show (i) the dispersion of the delay for each route size (scatter plot on top) and (ii) the distribution of route sizes (bottom). Additionally, we show the evolution of the distribution of delays for each route length for the real topology.

Random/balanced. The "random/balanced" topology connects routers in an Erdös-Rényi random graph [18], each router being linked to two other routers drawn at random. This topology has a low clustering, medium diameter and a balanced distribution of in-degrees. Figure 6.4 (a) shows a small dispersion of the delay for each route size (as expected from the small variation of delay per link) and most of the routes with lengths between 4 and 6 hops.

Scale-free network. The "scale-free" topology is representative of networks where central elements are acting as *hubs* in the network, e.g., the main routers in each linked institution on a campus. It uses a *preferential attachment* incremental construction (Albert & Barabási [4]). In this network, routers are added one after the other, with the target of their outgoing links selected as follows: a link from a router r targets a router r_d with a probability that is proportional to r_d's current in-degree. This means that the nodes that are more likely to increase their in-degree are those that already have a high in-degree. This topology presents a high clustering and a low diameter, and a sparse distribution of in-degrees. Figure 6.4 (b) shows the same small dispersion of the delay as before and, giving the higher clustering, shorter routes.

Small-world network. The "small-world" topology is built according to the shuffling model of Watts & Strogatz [93]: starting with a ring composed of all routers (each router being linked to its two neighbors in the ring), randomly chosen links are shuffled and directed to random routers, creating shortcuts. This topology presents a high clustering, low diameter and balanced in-degrees. Figure 6.4 (c) shows the small dispersion of the delay and slightly shorter routes than in the Random/balanced topology.

Random/sparse. The "random/sparse" topology is similar to the random/balanced one, which has a low clustering, medium diameter and a balanced distribution of in-degrees, but it has the particularity of having highly varying delays of 50 ms

-75%/+150% assigned to links. As can be seen in Figure 6.4 (d), this high variation affects both the dispersion of delays for a given route length, which is quite high comparing with the previous topologies, and the distribution of route size. The random/balanced and random/sparse topologies can be considered as the two extreme cases for this study. In the former, using delays as a metric for deciding on low-length routes is likely to succeed most often, while for the latter it is not, preferring the route length as metric.

Real topology (UIUC lab). Finally, our last topology, "UIUC laboratory", is part of a set of real Internet topologies [54] that were produced by collecting BGP routing maps and benchmarking inter-router delays. This topology corresponds to a large local area network on a university campus, composed of 448 pure routers and 203 routers and attachment points, connected by 8,486 links. In Figure 6.4 (e) we notice a small dispersion of the delay per route length at each one of three steps, which means that the routers are connected using three logical levels that use different delays. Also, the distribution of the route size is very wide, with routes of all lengths. Figure 6.4 (f) shows the delay distribution per route size using percentiles, that is, for a delay d and a route length of l, it shows the percentage of routes of length l that take d ms or less. To better explain the figure, we take as an example the routes of 6 hops in size. Only 5% of them have a delay of maximum 20 ms, while the median is at 60 ms (50% of these routes have a delay of maximum 60 ms). The maximum delay for the routes of length 6 does not overpass 120 ms. The small difference between the 5th and the 25th percentile, as well as between the 50th and the 75th percentile and the almost null difference between 90th percentile and the max delay show again this separation per logical levels (probably different ASes).

6.3.2 Experimental setup

Each experiment involves 10,000 nodes randomly distributed over routers. We measure the time used for propagation in *cycles*, with a cycle being the time period of all gossip-based protocols. As we assume that Vivaldi coordinates are bootstrapped (e.g., provided by an external service), we let the system run for 200 cycles to let them stabilize using a set of 20 landmarks nodes, before the actual gossip-based dissemination (push and pull) takes place. 500 messages from random initial peers are then published,

each of size 10kB, and the simulation stops when all peers have received all messages. We monitor the load on routers only during the dissemination phase. The parameters for the push dissemination are $f_{push} = 3$ (fanout) and $htl = 6$, which seeds 10% of the nodes. The dissemination is done synchronously with the cycles (a node that receives a message m forwards it to f_{push} other random peers during the next cycle, when it also sends 1 pull request).

6.3.3 Time efficiency of the dissemination

We first evaluate the impact of network-friendly gossiping strategies on the actual performance of the dissemination, i.e., if the number of cycles required to notify a given percentage of the network varies. We have conducted experiments for each topology using the three peer selection strategies (RR, CR and RC), where the close view is created using the route length metric.

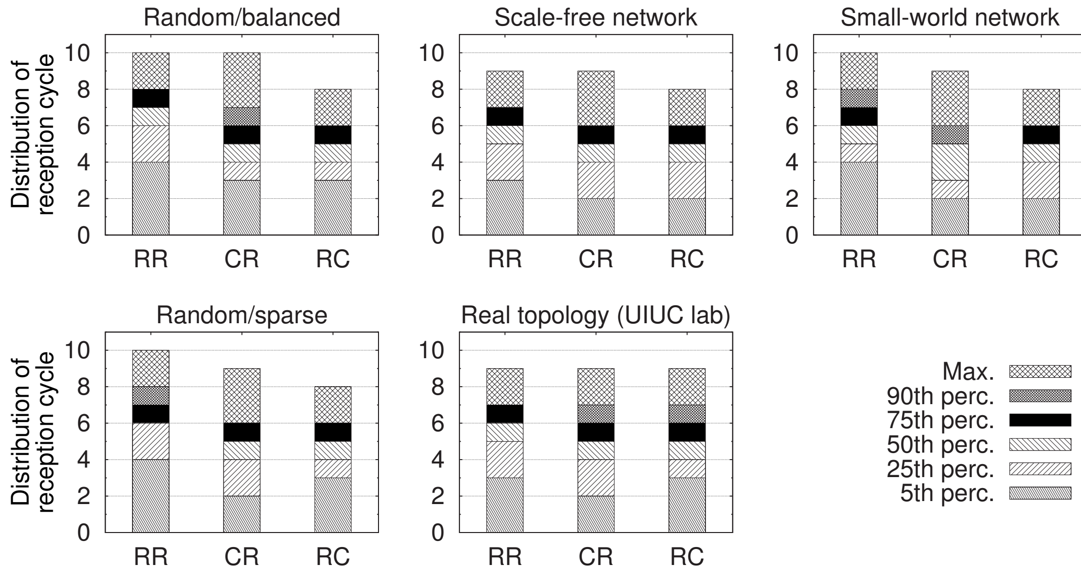


Figure 6.5: Distribution of the number of cycles required for receiving the *first* message.

Figure 6.5 shows the distribution of the number of cycles required to achieve complete dissemination by the means of percentiles. In all cases, the complete dissemination takes up to 8-10 cycles, and half of the nodes (the 50th percentiles is the median value)

receive the message within 4 or 5 cycles. We observe that using network-friendly gossiping even has a small positive impact, which is slightly more important when using RC. In the random/balanced topology, the complete dissemination is achieved in 10 cycles using RR or CR, and in 8 cycles for RC. Half of the nodes receive the epidemic message in 7 cycles for RR and in 5 cycles for CR and RC. The experiments run on all the other topologies show the same small positive impact. This conveys the fact that our protocols can reduce the load without affecting in a negative way the dissemination efficiency.

6.3.4 Impact on the load at each router

We evaluate the impact of our strategies on the load imposed on each router in the network, both in terms of number of messages and bandwidth. The first criterion is important as a longer path stresses more routers for every connection established and message sent along that path, while the second criterion represents the actual routing load at each router and is a fundamental concern to ensure true network friendliness. We run our experiments twice, for each of the two metrics.

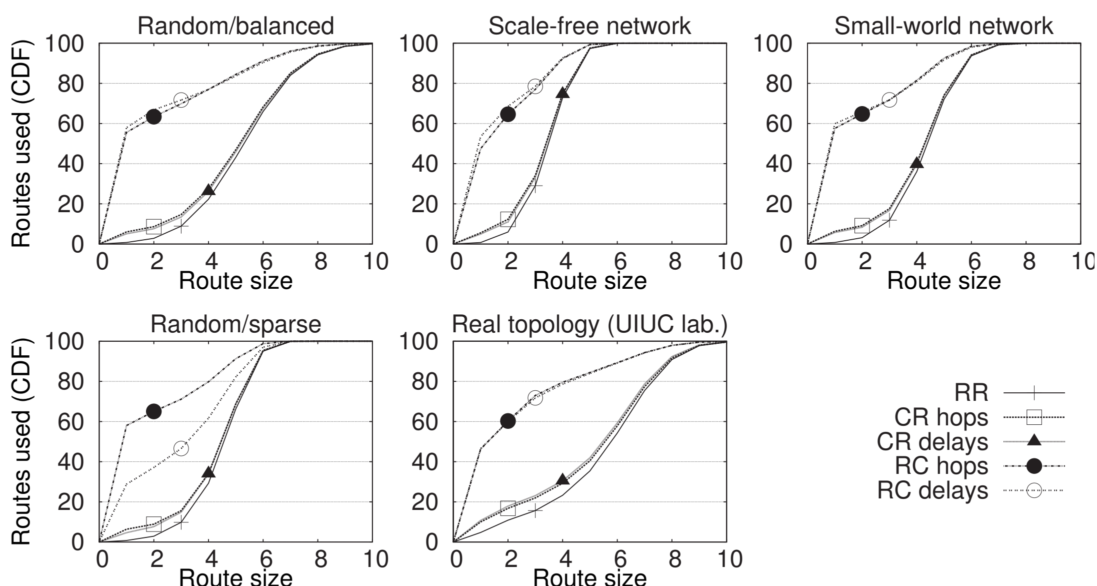


Figure 6.6: Distribution of load on all routers: route lengths.

Figure 6.6 presents the distribution of the route lengths, for all routes used during

one simulation, regardless of the size of the message. The push model is used only to seed the network and then the pull requests largely dominate until complete dissemination. As a consequence, it is expected to obtain roughly the same results for CR and RR, since they both use the random view for the pull requests. This is what we also see in the figure, and moreover using either metric. The CR and RR strategies use roughly routes of the same size, which means that these strategies produce nearly as much load in terms of number of messages on the routers. Conversely, the RC strategy, that uses the close view for the pull requests, uses routes of shorter size (already 60% of the used routes have only two hops for most topologies), which significantly reduces the load in the system. Different results in the usage of the two metrics can be seen only for the random/sparse topology, which is the synthetic topology with a mismatch between the delay and the route length; here, the number of hops yields better results than the delay.

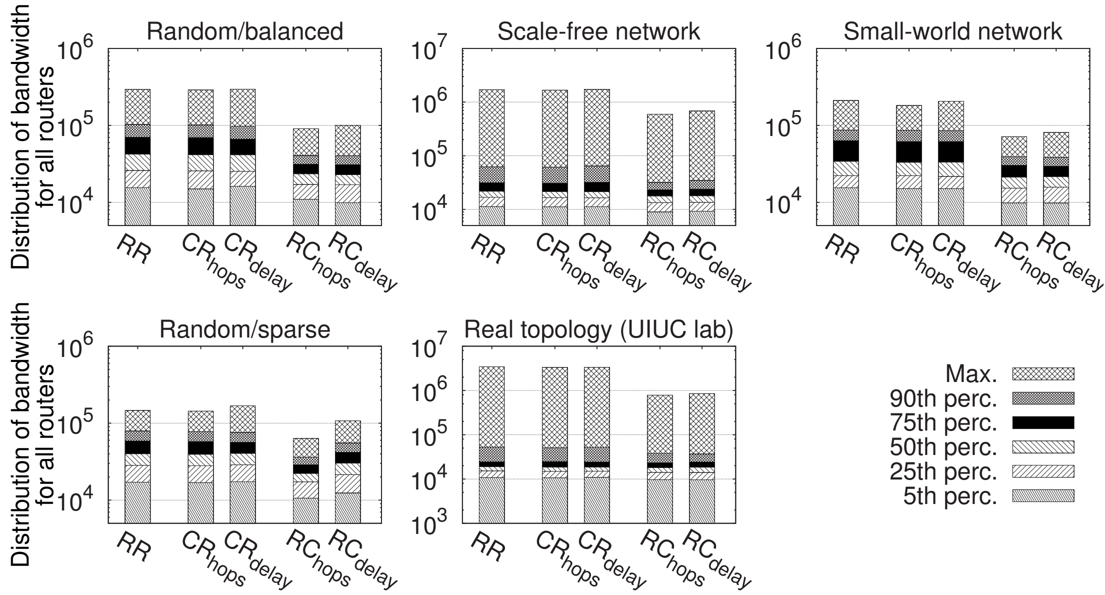


Figure 6.7: Distribution of load on all routers: bandwidth.

Figure 6.7 presents the distribution of bandwidth on all routers, for all messages sent during one simulation. 500 messages of size 10kB are sent, with pull requests and empty replies of 50 bytes. Note the logarithmic scale for the ordinates. We observe

on all topologies that the RC strategy greatly reduces the amount of data imposed on each router, with almost one order of magnitude. Moreover, it appears clearly that the CR strategy is not very efficient with respect to bandwidth, because most of the peers (all, except the 10% of peers from the initial seed) obtain the epidemic messages from random sources, which means that the routes that are used are not short. We also observe that RC is able to reduce not only the load on all routers, but also the difference between the median and the maximal load in all cases. Finally, in the random/sparse topology with the RC strategy, the load is much lower when using the number of hops as a metric rather than the delay, highlighting the benefits of the former metric for infrastructures where there is no clear matching between path lengths and delays. Note that this matching can be tested at runtime by the protocol itself, by comparing measures of path lengths and delays between random pairs of nodes, and switching to actual route length measurements when necessary.

6.3.5 Evaluation Review

The experiments conducted and presented in this section bring three main observations.

First, the infrastructure-awareness for gossip-based dissemination protocols, regardless the metric used for implementing it, *does not impact the performance* (delays and coverage) of the diffusion.

Second, the best policy for *ensuring the completion of the dissemination* with no or very few duplicates reception in an overall short dissemination delay is Random/Close (RC). The principle of RC is to seed the network by an initial set of random limited push operations (small values for fanout and hops-to-live), which limits the number of redundant epidemic messages, followed by pull operations that use *close* links for the majority of the exchanges. The dissemination is completed in 1 to 2 cycles less than for the other policies, except for the real topology, where the dissemination takes the same number of cycles.

Finally, the best results achieved for *reducing the load on the infrastructure* are obtained with the same RC policy. In the real topology, RC reduces up to five times the bandwidth when compared with the classical RR policy. Noteworthy, the load reduction that can be achieved by using application-level delays/RTT as a metric for constructing infrastructure-aware links is limited, and is depending on the correlation between path lengths and delays. This correlation is not necessarily present in real

networks or common synthetic network topologies. Therefore, the use of the measured route length as a metric for constructing close-links yields more stable and effective load reduction: e.g., in the Random/sparse topology, the bandwidth is 1.5 times lower when using the route length instead of the delay as a metric.

6.4 Summary and Discussion

In this chapter, we have presented a network-friendly approach for information dissemination using gossip-based protocols on multi-hops networks such as the Internet.

The primary challenge was to reduce the load volume at the underlay level using the classical gossip-based algorithms for information dissemination which are based on restricted flooding. As discussed before (in Chapter 3), load can be reduced through network proximity awareness: the same volume of overlay load is generating fewer load at the underlay level when overlay links are organised with proximity in mind. An important part of the burden usually imposed on the infrastructure can be thus avoided, and the remaining load can be better balanced amongst routers. Our approach mainly refers to clustering peers at short distances and to exploiting these short-distance links for communication.

The metrics used to cluster peers are delay and route length. Delay measurements have the advantage that are easily obtained at application level at very low costs, for example by computing the time until the reply to a request is received, knowing or not taking into consideration the time spent at the destination node. The delay gives the measurement in time to reach a peer from another, but it has the drawback that it may vary with the message size, and more importantly, it does not say much about the path that a message traverses. In any network and especially the Internet, there are links of different speed, so the number of routers that a message traverses in a certain delay, and thus the number of routers that increase their load, is uncertain.

For this reason, we have used as alternative metric the route length: when clustering peers, we choose peers that are reachable through short routes. The problem of the route length being a metric that is not easily obtained from the application level (being an information about the underlay that is considered to be unnecessary out of the underlay context) was overcome by our own mechanism to detect the path length

between two peers, which avoids using the traceroute tools and their security issues through a lightweight probing.

Through our evaluation experiments, we have observed that in topologies where there is not a direct correspondence between the delay and route length, as for the Internet or the random/sparse topology that we have studied before, the usage of delay as a metric reduces the load, but not as much as the route length metric. However, for topologies where the delays reflect the number of routers that are traversed, we have seen that the delay is an appropriate metric to use.

The metrics could go even further, by considering also the current or predicted load on each router. A peer A situated one router further than another peer B might be preferred as a neighbor for peer P if the path from P to A uses less loaded routers than the path from P to B. In such case, the communication would generate slightly more load, since more routers are involved, however, it would have the advantage of improving the load balancing.

We have also seen that the peer selection strategies play a substantial role in reducing the load. Among the three possible strategies, the best results are achieved by combining a limited push-based *seeding* of the network using random links, followed by periodic pull-based dissemination using short routes. This solution has the advantage of hardly generating duplicates when using the push model and most of the epidemic messages are transmitted through pull requests, between close-by peers. Moreover, its functionality requires a very low complexity.

Our RC strategy is very basic, which leaves place for improvements or other similar strategies, seemingly at the price of increased complexity. An idea could be to exploit more the push model. A simple but very small improvement could be for a peer that receives the message from a random peer to push it both to random neighbors and to close neighbors, but this would be almost equal to the RC strategy, since the close neighbors that receive the message through a push in this new strategy would anyway have obtained it quickly through a pull request in the RC strategy. A more complex and more compelling strategy could be to use some notion of direction and to disseminate the epidemic message in the network through pushes that at each step cover a different underlay distance. An epidemic message that is pushed at a distance that is halved from its previous step would intuitively yield a short dissemination time generating

only a small number of duplicates, however, it is not clear how this strategy would adapt to each topology and neither whether it would reduce the load as much as RC.

We can conclude that network friendliness has no impact on the efficiency of the gossip dissemination itself, making no tradeoff between the dissemination time and load reduction. Generalizing for any P2P communication, our network-friendly approach refers to reducing the load imposed on a system by an application, through proximity awareness: when the efficiency of the application can be kept the same regardless the size of the underlay routes, favouring the usage of short routes in terms of a specific metric significantly reduces the load in the underlay network.

Chapter 7

Conclusions

7.1 Summary

In this thesis, we have presented novel methods for managing the load in peer-to-peer systems. We have deal with load balancing and load reduction.

In Chapter 2 we have introduced peer-to-peer systems and presented the features of DHTs with an emphasis on the Chord, Pastry and CAN overlays. These overlays are the most related ones to our solutions presented in Chapters 4 and 5. We have also discussed the characteristics of gossip-based protocols as a means to self-organization. The mechanisms of gossiping have then been applied in our network-friendly information dissemination solution in Chapter 6.

A summary of related work was presented in Chapter 3. We have given an overview on existing load balancing solutions, identifying three main categories: object placement, traffic routing and load in the underlay. For each of them, we have presented the most relevant load balancing solutions, their attributes, advantages and drawbacks.

Our contributions were developed in Chapters 4, 5 and 6: a structured overlay with path redundancy (HyPeer), adaptive load balancing by link reorganization and network-friendly gossiping, respectively. We have dealt with (i) object and node placement for namespace balancing, (ii) routing tables (through link reorganization) and routing strategies for traffic load balancing and (iii) network awareness for reducing the traffic at the underlay level when disseminating information. We have assessed our solutions by way of thorough experiments. All our results have shown improved performance.

7.2 Contributions

Our mechanisms for load management were presented in the context of information lookup and information dissemination systems. They are based on a certain flexibility of the overlay, which allows either a large choice for the peers to be considered as neighbors, or several neighbors to be considered as candidates for a message transmission (as next hop for a request or partner peer in dissemination). Our solutions are simple and provide very good performance results.

As a first contribution, we proposed HyPeer, a structured overlay with flexible choice for the routing strategy to be used when forwarding requests. We manage the load through a balanced namespace and a routing strategy that balances the routing traffic. Our starting point was an analysis of a Chord-like system that uses greedy routing for forwarding its requests. Greedy routing is a simple routing strategy, however, under popular requests it generates path convergence and moreover, for fault tolerance, it lacks of dependability. Consequently, we designed both, the infrastructure and the routing strategies of our overlay with the goal to offer support for multiple routing strategies, that achieve load balancing, fault tolerance and also low path delay and short path length. We applied simple modifications to Chord-like systems: Chord cannot easily exploit redundant paths because of its non-determinism in node placement that does not permit treating digits in any order. We removed this constraint by adding some determinism in the placement of the nodes. This means that we obtained control on the position of the nodes on the ring, which is obviously advantageous for the routing strategy. In contrast to the common method of using a hash function to map the nodes on the ring, we approximated a hypercube structure by trying to maintain an even inter-node distance equal to a power of 2. [1] Then, we also modified the routing protocol to exploit alternative paths by taking into account all possible incoming links of the destination starting from the source node. The rate of request success is much higher, and the maintenance cost remains low, since no additional structures are required to be maintained. Our experiments clearly demonstrate that all routing strategies combined with uniform space partitioning provide the desired goal: short average path length, routing load balancing, fault tolerance and low average path delay.

[1] It was taken care that this inter-node distance can be maintained in case of churn.

Our second contribution concerns routing load balancing through adaptive *link reorganization* in DHT overlays. We presented an analysis of the load distribution in structured peer-to-peer systems taking into consideration the load caused by the popularity of the objects (Zipf-like requests). Based on this analysis, we proposed a novel approach to minimize the load generated by popular requests by reorganizing the routing tables accordingly. Our mechanisms neither require changes to the topology nor to the association rules (placement) of the objects to the peers. The updates in the routing tables with nodes that are less loaded send the requests on less loaded paths. The traffic redirection however cannot decrease the load below the number of requests addressed to a node. Thus, we still have a Zipf-like distribution of the load (request and routing load) on the peers but with much lower intensity. For this reason, we added the complementary solution of a caching mechanism to reduce also the request load. This is the only strategy in our solutions that requires some extra messages to be exchanged. Results from experimental evaluation demonstrate a more balanced traffic and, consequently, improved scalability and performance.

Last but not least, we proposed *network-friendly gossiping* for information dissemination. We analyzed the dissemination models and we proceeded on using a hybrid model by seeding the network through a push phase, then allowing complete dissemination through a pull phase. We used two views for peer selection in the gossip exchange: a random view, containing random neighbors from the system, and a close view, containing neighbors that are close in terms of a proximity metric. We used an application-level metric, the delay between any two peers, and a topology-level metric, the route length at the underlay level. The dissemination proved to be more efficient when the push and the pull models select partner peers from random and close neighbors, respectively. As metrics, the delay is not an appropriate metric for all types of topologies, especially for those that do not have a close correspondence between the delay and the number of hops between any two peers, while the route length, which can be determined by a lightweight probing method, can be used in any topology. The simulations on synthetic and real topologies have shown that the load in the underlay topology can be significantly reduced. Moreover, network friendliness does not have any negative impact on the dissemination itself.

7.3 Discussion and Outlook

Load management is an essential, distributed task in peer-to-peer systems. As a client, each peer generates some load on other peers, and as a server it becomes loaded through requests issued by other peers. Moreover, all system resources on the message paths get loaded by treating the messages. Overloaded peers cause slow response times while overloaded links affect all the traffic in the underlay network. Thus, where possible and appropriate, this load needs to be maintained low and balanced over participating resources.

In DHTs, the overlay construction is a very important phase for further load balancing, since the overlay structure is the foundation for the routing strategies. This starts with the placement of the nodes in the overlay. We have shown in Chapter 4 that a uniform deterministic placement of the nodes in the overlay (which usually gives more complex structures) is key for alternative paths, which can be randomly followed for routing load balancing. Peers should have a uniform inter-node distance not only for a balanced object load, but also to balance their in-degree under a uniform choice of the neighbors. The out-degree is most of the time balanced, by having all peers showing the same number of neighbors. This assures a minimum of routing load balancing in the system. However, under a non-uniform flow of requests, when a strict routing strategy is used (i.e., uses always the same routing table entry for the same requested key), some entries tend to be used much more than others. We have identified several solutions to overcome this problem. When the routing strategy is strict, we update the routing table entries: neighbors are replaced by less loaded peers (Chapter 5). When the routing strategy is itself flexible (i.e., uses different entries for the same requested key), we have proposed a random selection of these entries in LB-HyPeer (Chapter 4). None of these solutions increase significantly the average path length, making them applicable even preventive, when no load problem exists. When several routing strategies can be used, as in HyPeer, a random selection of these strategies at each peer in the path (thus a random selection from the corresponding routing table entries) would also intuitively provide a certain level of routing load balancing.

An example of an overlay with flexible choice of the neighbors for a peer is Pastry, while an example of an overlay with a flexible choice of the neighbor to be used to send a message is CAN. However, there are overlays that are less or not flexible. Chord is

an example of a less flexible overlay. In Chord, one could imagine applying a routing strategy that covers first the small hops and then larger ones, as we do in HyPeer, however, due to the lack of precision in the position of the neighbors, the routing path might become very long. This would generate more load in the system, having more peers on the path. This solution being thus not efficient in this case, we have proposed HyPeer. Its advantage is that the average path length in its routing strategies does not vary much. This is valid for the four analyzed routing strategies, but as a consequence also in any other new routing strategy that uses the same principle of hops of powers of 2. Thus, as future directions, new routing strategies can be developed. They may cover other aspects, such as security issues: for example, when choosing the next hop for a request, the candidate neighbors can be sorted based on their levels of trust as peers or as participants in a specific application.

We have addressed the underlying topology in both, DHTs and information dissemination systems. In order to direct the overlay traffic such that the underlay traffic is reduced, some knowledge of the underlying topology is mandatory. Proximity awareness can be successfully applied to DHTs, as with the Pr-HyPeer routing strategy, which uses proximity routing to forward requests in order to achieve a low average path delay. The Pr-HyPeer routing strategy is as such independent of the metric. Therefore, the analysis of the metrics to be used in order to efficiently reduce the load at the underlay level still needs to be explored. However, through our analysis of two metrics (application-level and topology-level) in Chapter 6, we have shown that the choice of the metric can highly influence the quantity of traffic load at the underlay level. This happens especially when the underlay topology has not a close correspondence between the delay and the number of hops between any two peers. A message generates a load on each router that it traverses, so the metric should reflect this total load. Thus, it is more appropriate to consider as a close neighbor a peer that is reachable through a low number of routers. Alternatively, when the correspondence is close, application-level metrics (the delay between two peers) is a much more handy metric.

In our information dissemination solution, we reduce the traffic at the underlying level by a dissemination mechanism that uses a random push for the initial seed and close pull for complete dissemination. These are rather simple, yet effective, selection strategies. Still to be investigated, combinations of these two strategies or a push with

more sense of direction (not random), could most likely reduce even more the routing load.

In this overlay, the choice of the random neighbors is flexible, while the close neighbors are chosen based on proximity metrics. No other constraints are imposed on the peers in order to become neighbors. However, there are some applications and systems that have specific constraints. Even though we do not consider them in this study, mobile systems are a good example of system with constraints, which are related to the physical position of the peers. However, even in this case, the principle of the dissemination would remain the same. We propose to analyze the possibility of replacing the random push with random walk to seed the system. Moreover, in dissemination systems where the information needs to be sent only to a part of the peers, the gossiping protocols would create a new overlay, containing only that part of the peers, making our solution applicable with no modifications.

Publications

[1] S. Bianchi, S. Serbu, P. Felber, and P. Kropf. **Adaptive Load Balancing for DHT Lookups**. In *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN)*, pages 411–418, 2006.

[2] M. Brogle, S. Serbu, D. Milic, P. H. M. Anwander and, C. Spielvogel, C. Fautsch, D. Harmanci, L. Charles, H. Sturzrehm, G. Wagenknecht, T. Braun, T. Staub, C. Latze, and R. Standtke. **BeNeFri Summer School 2009 on Dependable Systems (Technical Report)**. Technical report, Münchenwiler, Switzerland, 2009. IAM-09-006.

[3] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. **Dynamic Load Sharing in Peer-to-Peer Systems: When some Peers are more Equal than Others**. In *Proceedings of the 2006 Montreal Conference on eTechnologies (MCETECH)*, pages 149–156, 2006.

[4] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. **Dynamic Load Sharing in Peer-to-Peer Systems: When Some Peers Are More Equal than Others**. *IEEE Internet Computing*, **11**(4):53–61, 2007.

[5] S. Serbu, P. Felber, and P. Kropf. **HyPeer: Structured Overlay with Flexible-Choice Routing**. Submitted to Elsevier Computer Networks Journal (ComNet), 2010.

[6] S. Serbu, P. Kropf, and P. Felber. **Fault-Tolerant P2P Networks: How Dependable is Greedy Routing?** In *Workshop on Dependable Application Support in Self-Organising Networks (DASSON)*, 2007.

[7] S. Serbu, P. Kropf, and P. Felber. **Improving the Dependability of Prefix-Based Routing in DHTs**. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA,*

and IS, number 4803 in Lecture Notes in Computer Science, pages 206–225. Springer Berlin / Heidelberg, 2007.

[8] S. Serbu, É. Rivière, and P. Felber. **Network-Friendly Gossiping**. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2009.

[9] C. Spielvogel, S. Serbu, P. Felber, and P. Kropf. **Semantic based Error Avoidance and Error Correction for Video Streaming**. In *Semantics in Adaptive and Personalized Services, Methods Tools and Applications.* ISBN: 978-3-642-11683-4.

[10] C. Spielvogel, S. Serbu, P. Felber, and P. Kropf. **A Model for Error Avoidance and Error Correction in Peer-to-Peer Networks**. In *Proceedings of the 2008 Third International Workshop on Semantic Media Adaptation and Personalization (SMAP)*, pages 100–105, Washington, DC, USA, 2008. IEEE Computer Society.

Bibliography

[1] V. Aggarwal, O. Akonjang, and A. Feldmann. **Improving User and ISP Experience through ISP-aided P2P Locality**. In *Proceedings of 11th IEEE Global Internet Symposium (GI)*. IEEE Computer Society, Washington, DC, USA, 2008. 58, 61, 113, 115

[2] J. I. Alvarez-Hamelin, A. C. Viana, and M. D. Amorim. **DHT-based Functionalities Using Hypercubes**. In *Proceedings of World Computer Congress (IFIP WCC)*, **212**, pages 157–176, 2006. 54, 55, 60

[3] N. Ball and P. Pietzuch. **Distributed Content Delivery using Load-Aware Network Coordinates**. In *Proceedings of the 3rd International Workshop on Real Overlays and Distributed Systems (ROADS)*, 2008. 113

[4] A.-L. Barabasi and R. Albert. **Emergence of Scaling in Random Networks**. *Science*, **286**, 1999. 124

[5] S. Bianchi, S. Serbu, P. Felber, and P. Kropf. **Adaptive Load Balancing for DHT Lookups**. In *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN)*, pages 411–418, 2006. 46, 47, 48, 49, 52, 60, 89

[6] M. Bienkowski, M. Korzeniowski, and F. M. a. d Heide. **Dynamic Load Balancing in Distributed Hash Tables**. In *Lecture Notes in Computer Science*, **3640**, pages 217–225. Springer Berlin / Heidelberg, 2005. 40, 60

[7] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. **Bimodal Multicast**. *Transactions on Computer Systems (TOCS)*, **17**(2):41–88, 1999. 25, 26, 120

[8] R. Braden. **Requirements for Internet Hosts: Communication Layers**. Internet Engineering Task Force RFC 1122, oct 1989. 115

[9] L. Breslau, P. Cao, G. P. L. Fan, and S. Shenker. **Web Caching and Zipf-like Distributions: Evidence and Implications**. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 126–134, 1999. 91, 107

[10] J. Byers, J. Considine, and M. Mitzenmacher. **Simple Load Balancing for Distributed Hash Tables**. In *Lecture Notes in Computer Science*, **2735**, pages 80–87. Springer Berlin / Heidelberg, 2003. 44, 60

[11] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. **Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks**. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002. 56, 61

[12] E. Cohen and S. Shenker. **Replication Strategies in Unstructured Peer-to-Peer Networks**. In *In Proceedings of ACM Special Interest Group on Data Communications (SIGCOMM)*, August 2002. 46, 49, 60

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. **Vivaldi: A Decentralized Network Coordinate System**. In *Proceedings of the ACM Special Interest Group on Data Communications (SIGCOMM)*, pages 15–26, 2004. 58, 61, 117, 120

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. **Wide-Area Cooperative Storage with CFS**. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–215, 2001. 40, 60

[15] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. **Towards a Common API for Structured Peer-to-Peer Overlays**. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, February 2003. 7

[16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. **Epidemic Algorithms for Replicated Database Maintenance**. In *6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12. ACM New York, NY, USA, 1987. 21

[17] N. Efthymiopoulos, A. Christakidis, S. Denazis, and O. Koufopavlou. **Enabling Locality in a Balanced Peer-to-Peer Overlay**. In *IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1–5, 2006. 58, 61

[18] P. Erdös and A. Rényi. **On the Evolution of Random Graphs**. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, **5**:17–61, 1960. 23, 124

[19] P. T. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. **Lightweight Probabilistic Broadcast**. *ACM Transactions on Computer Systems (TOCS)*, **21**(4), 2003. 25, 120

[20] Y. Fernandess and D. Malkhi. **On Collaborative Content Distribution using Multi-Message Gossip**. *Journal of Parallel and Distributed Computing (JPDC)*, **67**(12):1232–1239, 2007. 28

[21] `http://freenetproject.org/`. 11

[22] A. Ganesh, A. Kermarrec, and L. Massoulie. **Peer-to-Peer Membership Management for Gossip-based Protocols**, 2003. 27

[23] L. Garces-Erice, K. W. Ross, E. W. Biersack, P. A. Felber, and G. Urvoy-Keller. **Topology-Centric Look-Up Service**. In *COST264/ACM 5th International Workshop on Networked Group Communications (NGC)*, pages 58–69. Springer, 2003. 33, 34, 57, 61

[24] D. Gavidia, S. Voulgaris, and M. van Steen. **A Gossip-based Distributed News Service for Wireless Mesh Networks**. In *Proceedings 3rd IEEE Conference on Wireless On demand Network Systems and Services (WONS)*, 2006. 26

[25] `http://rfc-gnutella.sourceforge.net/`. 11

[26] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. **Load Balancing in Dynamic Structured P2P Systems**. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Hong Kong, 2004. 32, 37, 41, 43, 60, 102

[27] P. B. Godfrey and I. Stoica. **Heterogeneity and Load Balance in Distributed Hash Tables**. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2005. 11, 39, 41, 43, 60

[28] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. **Adaptive Replication in Peer-to-Peer Systems**. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 360–369. IEEE Computer Society, 2004. 46, 47, 60

[29] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. **Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload**. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 314–329, 2003. 92

[30] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. **The Impact of DHT Routing Geometry on Resilience and Proximity**. In *Proceedings of ACM Special Interest Group on Data Communications (SIGCOMM)*, pages 381–394, 2003. 91

[31] A. Gupta, P. Dinda, and F. E. Bustamante. **Distributed Popularity Indices**. In *Proceedings of ACM Special Interest Group on Data Communications (SIGCOMM)*, 2005. 89, 91

[32] S. C. Han and Y. Xia. **Network Load-Aware Content Distribution in Overlay Networks**. *Computer Communications*, **32**(1):51–61, 2009. 59, 61

[33] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. **Locating internet bottlenecks: algorithms, measurements, and implications**. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 41–54, New York, NY, USA, 2004. ACM. 116

[34] K.-L. Huang, T.-Y. Huang, and J. C. Y. Chou. **LessLog: A Logless File Replication Algorithm for Peer-to-Peer Distributed Systems**. *18th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, **1**:82b, 2004. 46, 48, 60

[35] K. Iwanicki, M. van Steen, and S. Voulgaris. **Gossip-Based Clock Synchronization for Large Decentralized Systems**. In *Proceedings of the 2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan)*, pages 28–42, Dublin, Ireland, June 2006. Springer-Verlag LNCS 3996. 28

[36] M. Jelasity and O. Babaoglu. **T-Man: Gossip-based Overlay Topology Management**. In *Proceedings of Engineering Self-Organising Applications (ESOA)*, Jul 2005. xv, 21, 22, 24, 25, 120

[37] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. v Steen. **The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations**. In *Middleware*, pages 79–98, 2004. 23

[38] M. Jelasity and A.-M. Kermarrec. **Ordered Slicing of Very Large-Scale Overlay Networks**. In *Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 117–124, Washington, DC, USA, 2006. IEEE Computer Society. 22, 27

[39] M. Jelasity, W. Kowalczyk, and M. v Steen. **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)*. IEEE Computer Society, 2004. 27

[40] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. **Gossip-Based Peer Sampling**. *TOCS*, **25**(3):8, 2007. 23

[41] M. F. Kaashoek and D. R. Karger. **Koorde: A Simple Degree-Optimal Distributed Hash Table**. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, pages 323–336, 2003. 13, 51

[42] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. **Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web**. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997. 39, 59, 60

[43] D. R. Karger and M. Ruhl. **Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems**. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 36–43, New York, NY, USA, 2004. ACM. 42, 43, 60

[44] R. M. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. **Randomized Rumor Spreading**. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 565–574, 2000. 25, 120

[45] S. Kashyap, S. Deb, K. V. M. Naidu, R. Rastogi, and A. Srinivasan. **Efficient gossip-based aggregate computation**. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 308–317, New York, NY, USA, 2006. ACM. 22, 27

[46] D. Kempe, A. Dobra, and J. Gehrke. **Gossip-Based Computation of Aggregate Information**. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, Washington, DC, USA, 2003. IEEE Computer Society. 27

[47] K. Kenthapadi and G. S. Manku. **Decentralized algorithms using both local and random probes for P2P load balancing**. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, New York, NY, USA, 2005. ACM. 33, 40, 60

[48] A.-M. Kermarrec, L. Massouli, and A. J. Ganesh. **Probabilistic Reliable Dissemination in Large-Scale Systems**. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, **14**:248–258, 2003. 25

[49] A.-M. Kermarrec and M. van Steen, editors. *ACM SIGOPS OSR, s.i. on Gossip-Based Computer Networking*, **41:5**. ACM, oct 2007. 22

[50] A. Klemm, C. Lindemann, M. K. Vernon, and O. P. Waldhorst. **Characterizing the Query Behavior in Peer-to-Peer File Sharing Systems**. In *Proceedings of ACM Internet Measurement Conference*, 2004. 92, 107

[51] T. Kojima, M. Asahara, K. Kono, and H. A. **Embedding Network Coordinates into the Heart of Distributed Hash Tables**. In *IEEE 9th International Conference on Peer-to-Peer Computing (P2P)*, pages 155 – 158, Seattle, WA, 2009. IEEE Computer Society. 57

[52] D. Kostoulas, D. Psaltoulis, I. Gupta, K. P. Birman, and A. J. Demers. **Active and Passive Techniques for Group Size Estimation in Large-Scale and Dynamic Distributed Systems**. *Journal of Systems and Software*, **80**(10):1639–1658, 2007. 27, 119

[53] J. Ledlie and M. Seltzer. **Distributed, Secure Load Balancing with Skew, Heterogeneity and Churn**. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, **2**, pages 1419–1430. IEEE, 2005. 42, 43, 60

[54] M. Liljenstam, J. Liu, and D. M. Nicol. **Development of an Internet Backbone Topology for Large-Scale Network Simulations**. In *Proceedings of the 2003 Winter Simulation Conference*, pages 694–702, 2003. 125

[55] T. Locher, S. Schmid, and R. Wattenhofer. **eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System**. In *Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 3–11, Washington, DC, USA, 2006. IEEE Computer Society. 51, 54, 60

[56] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. **Search and Replication in Unstructured Peer-to-Peer Networks**. In *Proceedings of the 16th international conference on Supercomputing (ICS)*, pages 84–95, New York, NY, USA, 2002. ACM. 47, 60, 97

[57] D. Malkhi, M. Naor, and D. Ratajczak. **Viceroy: A Scalable and Dynamic Emulation of the Butterfly**. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, 2002. 13, 51

[58] B. Maniymaran, M. Bertier, and A.-M. Kermarrec. **Build One, Get One Free: Leveraging the Coexistence of Multiple P2P Overlay Networks**. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS)*, June 2007. 22

[59] P. Maymounkov and D. Mazieres. **Kademlia: A Peer-to-Peer Information System Based on the XOR Metric**. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002. 13, 51

[60] A. Montresor, M. Jelasity, and O. Babaoglu. **Chord on Demand**. In *Proceedings of the 5th International Conference on Peer-to-Peer Computing (P2P)*, pages 87–94, Konstanz, Germany, August 2005. IEEE. 24

[61] D. Mosk-Aoyama and D. Shah. **Computing separable functions via gossip**. In *Proceedings of the 25th annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 113–122, New York, NY, USA, 2006. ACM. 22

[62] J. A. Patel, É. Rivière, I. Gupta, and A.-M. Kermarrec. **Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems**. *Computer Networks*, **53**(13):2304–2320, 2009. Gossiping in Distributed Systems. 7

[63] `http://www.planet-lab.org/`. 8, 116

[64] C. G. Plaxton, R. Rajaraman, and A. W. Richa. **Accessing Nearby Copies of Replicated Objects in a Distributed Environment**. In *Proceedings of the 9th annual ACM symposium on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, New York, NY, USA, 1997. ACM. 48, 59

[65] V. Ramasubramanian and E. G. Sirer. **Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays**. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*, pages 8–8. USENIX Association, 2004. 46, 48, 60

[66] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. **Load Balancing in Structured P2P Systems**. In *Lecture Notes in Computer Science*, **2735**, pages 68–79. Springer Berlin / Heidelberg, 2003. 40, 43, 60

[67] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. **A Scalable Content Addressable Network**. In *Proceedings of ACM Special Interest Group on Data Communications (SIGCOMM)*, pages 161–172, 2001. xv, 13, 14, 18, 19, 45, 51, 55, 57, 60, 61

[68] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. **Topologically-Aware Overlay Construction and Server Selection**. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, June 2002. 34

[69] É. Rivière. *Collaborative Overlay Networks for Decentralized Search in Large-Scale Distributed Systems*. PhD dissertation, University of Rennes 1, November 2007. 22

[70] M. Roussopoulos and M. Baker. **Practical Load Balancing for Content Requests in Peer-to-Peer Networks**. *Distributed Computing*, **18**(6):421–434, 2002. 49, 60

[71] A. Rowstron and P. Druschel. **Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems**. In R. Guerraoui, editor, *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, number 2218 in Lecture Notes in Computer Science, pages 329–350. Springer Heidelberg, Germany, 2001. xv, 13, 14, 15, 17, 34, 51, 53, 112, 115

[72] A. Rowstron and P. Druschel. **Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility**. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 188–201, 2001. 48

[73] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. **HyperCuP – Hypercubes, Ontologies and Efficient Search on P2P Networks**. *Lecture Notes in Computer Science*, **2530**:133–134, 2002. 54, 60

[74] S. Seetharaman, V. Hilt, M. Hofmann, and M. Ammar. **Preemptive Strategies to Improve Routing Performance of Native and Overlay Layers**. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM)*, 2007. 58, 61, 113

[75] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. **Dynamic Load Sharing in Peer-to-Peer Systems: When some Peers are more Equal than Others**. In *Proceedings of the 2006 Montreal Conference on eTechnologies (MCETECH)*, pages 149–156, 2006. 52, 60, 89

[76] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. **Dynamic Load Sharing in Peer-to-Peer Systems: When Some Peers Are More Equal than Others**. *IEEE Internet Computing*, **11**(4):53–61, 2007. 32, 89

[77] S. Serbu, P. Felber, and P. Kropf. **HyPeer: Structured Overlay with Flexible-Choice Routing**. Submitted to Elsevier Computer Networks Journal (ComNet), 2010. 54, 57, 60, 63

[78] S. Serbu, P. Kropf, and P. Felber. **Fault-Tolerant P2P Networks: How Dependable is Greedy Routing?** In *Workshop on Dependable Application Support in Self-Organising Networks (DASSON)*, 2007. 13, 50, 63

[79] S. Serbu, P. Kropf, and P. Felber. **Improving the Dependability of Prefix-Based Routing in DHTs**. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, number 4803 in Lecture Notes in Computer Science, pages 206–225. Springer Berlin / Heidelberg, 2007. 33, 34, 63

[80] S. Serbu, É. Rivière, and P. Felber. **Network-Friendly Gossiping**. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2009. 59, 61, 111

[81] H. Shen. **An Efficient and Adaptive Decentralized File Replication Algorithm in P2P File Sharing Systems (EAD)**. In *Eighth International Conference on Peer-to-Peer Computing*, 2008. 46, 49, 60

[82] H. Shen and C.-Z. Xu. **Locality-Aware Randomized Load Balancing Algorithms for DHT Networks**. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP)*, pages 529–536, Washington, DC, USA, 2005. IEEE Computer Society. 40, 57

[83] H. Shen and Y. Zhu. **Plover: A Proactive Low-Overhead File Replication Scheme for Structured P2P Systems**. In *IEEE International Conference on Communications (ICC)*, pages 5619–5623, 2008. 33, 46, 49, 56, 60, 61

[84] K. Sripanidkulchai. **The Popularity of Gnutella Queries and its Implications on Scalability**. White paper, Carnegie Mellon University, 2001. 92

[85] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. **Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications**. In *Proceedings of ACM Special Interest Group on Data Communications (SIGCOMM)*, pages 149–160, 2001. 13, 14, 15, 51, 53, 63, 65

[86] R. van Renesse, Y. Minsky, and M. Hayden. **A Gossip-Style Failure Detection Service**. Technical Report TR98-1687, Cornell University, 1998. 22

[87] S. Voulgaris, D. Gavidia, and M. van Steen. **CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays.** *Journal of Parallel and Distributed Computing (JPDC)*, **13**(2), 2005. 21, 22, 23, 120

[88] S. Voulgaris, M. Jelasity, and M. van Steen. **A Robust and Scalable Peer-to-Peer Gossiping Protocol**. In *Proceedings of the 2nd International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, pages 47–58, 2003. 26

[89] S. Voulgaris and M. van Steen. **An Epidemic Protocol for Managing Routing Tables in Very Large Peer-to-Peer Networks**. In *Lecture Notes in Computer Science: Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, **2867**, pages 41–54. Springer, 2003. 26

[90] S. Voulgaris and M. v Steen. **Epidemic-Style Management of Semantic Overlays for Content-Based Searching**. In *Euro-Par*, pages 1143–1152, 2005. 24

[91] C. Wang, L. Xiao, Y. Liu, and P. Zheng. **DiCAS: An Efficient Distributed Caching Mechanism for P2P Systems**. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, **17**(10):1097–1109, 2006. 46, 48, 60

[92] Q. Wang, K. Daudjee, and M. T. Özsu. **Popularity-Aware Prefetch in P2P Range Caching**. In *Proceedings of the 8th International Conference on Peer-to-Peer Computing (P2P)*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society. 47, 60

[93] D. J. Watts and S. H. Strogatz. **Collective Dynamics of 'Small-World' Networks.** *Nature*, **393**(6684):440–442, June 1998. 124

[94] Y. Xia, A. Dobra, and S. C. Han. **Multiple-Choice Random Network for Server Load Balancing**. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM)*, pages 1982–1990, 2007. 46, 47, 48, 60

[95] H. Yamamoto, D. Maruta, and Y. Oie. **Replication Methods for Load Balancing on Distributed Storages in P2P Networks**. *IEICE Transactions*, **89-D**(1):171–180, 2006. 34, 47, 49, 60

[96] M. Zaharia and S. Keshav. **Gossip-based Search Selection in Hybrid Peer-to-Peer Networks**. In *International workshop on Peer-To-Peer Systems (IPTPS)*, 2006. 28

[97] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. **Tapestry: A Resilient Global-scale Overlay for Service Deployment**. *IEEE Journal on Selected Areas in Communications (JSAC)*, **22**(1):41–53, 2004. 13, 51

[98] Y. Zhu and Y. Hu. **Efficient, Proximity-Aware Load Balancing for Structured P2P Systems**. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P)*, page 220, Washington, DC, USA, 2003. IEEE Computer Society. 57