Beernet: A Relaxed Approach to the Design of Scalable Systems with Self-Managing Behaviour and Transactional Robust Storage

Boris Mejías Candia

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor in Engineering Sciences

October, 2010

ICTEAM, pôle d'Ingéniérie Informatique
École Polytechnique de Louvain
Université catholique de Louvain
Louvain-la-Neuve
Belgium

I am glad to be able to tell you in front of all my guests – despite the fact that their presence here is proof to the contrary – that your theory is intelligent and sound.

"The Master and Margarita" - Mikhail Bulgakov

To Saartje and Ina

Abstract

Distributed systems are becoming larger, more dynamic, more complex and therefore, difficult to manage. Imposing strict requirements on the underlying distributed system becomes counterproductive and inconvenient, specially when those requirements are hard to meet. This work is about relaxing conditions and requirements to cope with the inherent concurrency and asynchrony of distributed systems, without sacrificing functionality. We develop through this dissertation the idea of relaxing conditions as a design philosophy: the relaxed approach. We design algorithms according to this philosophy to build Beernet, a scalable and self-managing decentralized system with transactional robust storage.

Beernet relaxes the ring structure for Chord-like peer-to-peer networks to not rely on transitive connectivity nor on perfect failure detection. The relaxed ring improves lookup consistency without sacrificing efficient routing. To provide transactional replicated storage, Beernet uses the Paxos consensus algorithm, where only the majority of replicas needs to be updated, instead of all of them as in traditional databases. We also relax ordering and versioning in data collections to provide concurrent access to them without locking the collections, hence improving performance. Relaxing these conditions on data replication, Beernet is able to provide transactional support, scalable storage and fault-tolerance, without sacrificing strong consistency.

Beernet stands for peer-to-peer and beer-to-beer network. Because beer is a known means to achieve relaxation, its name reflects the design philosophy behind it.

Ubuntu - Acknowledgements

Ubuntu is a humanist philosophy originated in Africa that says that “I am what I am because of who we all are.” I believe in such philosophy, and therefore, these acknowledgements make a lot of sense to me, because this thesis is what it is thanks to the contribution of many people.

I would like to start by expressing my gratitude to my promoter Peter Van Roy, who believed that I could make a contribution despite the slow start I had. Working with Peter has been an excellent experience, specially because he gives importance not only to the scientific reasoning, but also to the development of systems to validate ideas. I also appreciate his will to hear and discuss my own ideas, and for his effort to understand my sense of humor.

I am also grateful to the jury members for their constructive comments on my dissertation. They really helped me to improve my thesis and the way I presented the results. I would like to specially thank Erik Klintskog for the time he spent talking to me from the beginning of my work. His honest and pragmatic way of approaching life and research made an impact on me.

My current and former colleagues from the distoz research group has also contributed directly to this work. Gustavo Gutiérrez, Yves Jaradin, Jérémie Melchior and David Calomme has been very important to the development of Beernet. The first version of the relaxed ring was conceived working with Donatien Grolaux, and inspired by the work of Kevin Glynn. Kevin has also helped me a lot to understand concepts of functional and distributed programming, together with the rules of cricket. I am also grateful to Luis Quesada, Fred Spiessens, Valentin Mesaros, Bruno Carton, Stefano Gualandi, James Ortiz and Raphaël Collet, with whom I had interesting discussions about the fundamental concepts of distributed programming.

But I am also thankful to researchers working in other research groups. Alfredo Cádiz was a regular co-author with whom I worked on the ideas of PALTA, Beernet's architecture and key/value-sets. John Ardelius was a very important collaborator in the work of understanding the influence of NAT devices. Trappist is what it is thanks to Monika Moser, Mikael Högqvist and other researchers from the SELFMAN project, specially from ZIB and KTH. Jorge Vallejos contributed to the understanding feedback loops. Eric Tanter helped me to present my work several times in Chile. John Thomson and Paulo Trezentos has contributed to key/value-sets. And of course, I am grateful to Sebastián González, who has always been there to help me understand whatever I needed to understand. He has been a very close friend during all this grad years. I also would like to thank people who read part of this dissertation and gave me important comments: Alfredo, Sebastián, Jorge, Ako and others.

I would like to acknowledge the European projects that have supported this research, specially SELFMAN and MANCOOSI. Large part of this work was developed in the context of these projects. I also got support from EVERGROW, MILOS, CoreGRID, and the Université catholique de Louvain, which gave me a two years position as teaching assistant.

The INGI department has been an excellent working pole. I am glad I could do my PhD here. Special thanks to the INGI running team, with Christophe, Sebastián, Pierre and all temporary members. Life at INGI would have not been the same without the comrades of Sindaca: Diego, Tania, Gustavo, Alfredo, Sebastián, Sergio, Lili, Nicolás, Ángela, Benjamin, Sandrine, James, Raph, Silvia, Nizar, Juan Pablo, Daniel and all other members. Also very important in this period of my life have been Manzana Mecánica, The Confused Deputies, Free Runners and Sharks Tongeren, and of course, all my friends. You known that even if your name is not listed here, I will always thank you for being there.

I got motivated to do research thanks to my experience at the Prog Lab, VUB, working with Wolfgang De Meuter during the EMOOSE program. I am really thankful to Wolfgang, Isabel, Johan, Theo and the rest of Prog for triggering my will to do a PhD.

I will take the freedom to thank the 33 miners currently trapped in a mine in Copiapo, and the people who suffered the earthquake in Chile this year. They helped indirectly by motivating me to finish my thesis with their example.

I am very lucky to have an incredible family in Chile and Belgium. Lia, Marc, Kristien, Peter, Daan and Kaat have welcomed me as if they would have known about Ubuntu already. Jorge, Mary and MoN has been closer than what they think. The distance means nothing with them. I am really grateful to their support, love and generosity. Papá, mamá y hermana, les amo mucho.

Saartje and Ina, my deepest gratitude to both of you. I never got tired of my thesis because you two provided me with the best break I could have. I enjoyed this period of my life because I could share it with you. Ina, gracias por tu alegría. Saartje, compañera de mis días y del porvenir, thanks for everything. This work is dedicated to you two.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

'Drink up.'
'Why three pints all of a sudden?'
'Muscle relaxant, you'll need it.'

"The Hitchhiker's Guide to the Galaxy" - Douglas Adams

There are many technological and social factors that make peer-to-peer systems a popular way of conceiving distributed systems nowadays. From the social point of view, people are willing to be more connected to other people, and to share their resources to get something back. From the technological point of view, the increment of network bandwidth and computing power has definitely made an impact on distributed systems which are becoming larger, more complex and therefore, difficult to manage. Imposing strict requirements on the underlying distributed system to deal with this complexity becomes counterproductive and inconvenient, specially when those requirements are hard to meet. This work is about relaxing conditions and requirements to cope with the inherent concurrency and asynchrony of distributed systems, without sacrificing functionality. We develop through this dissertation the idea of relaxing conditions as a design philosophy: the relaxed approach. We design algorithms according to this philosophy to build Beernet, a scalable and self-managing decentralized system with transactional robust storage.

As first example, we take the classical client-server architecture, which provides a simple management scheme with centralized control of the whole system. But it does not scale because the server becomes a point of congestion and a single point of failure. If the server fails, the whole system collapses. This implies too much requirements on the server. Decentralized systems relax this condition making that every node in the system becomes a server, so that every node plays the role of client and server at the same time. This strategy reduces congestion and eliminates the single point of failure. Of course, decentralization introduces new challenges that will be discussed along this dissertation.

An important concept to deal with the complexity of large-scale decentralized systems is to make them self-managing. Peer-to-peer networks, and especially in their form of structured overlays, offer a fully decentralized architecture which is often self-organizing and self-healing. These properties are very important to build systems that are more complex than file-sharing, which is currently the most common use of peer-to-peer. Structured overlay networks (SONs) are networks built on top of another one, using a structure to organize the participant nodes, instead of connecting them randomly. Despite the nice design of many existing SONs, many of them present problems when they are implemented in real-case scenarios. The problems arise due to basic issues in distributed computing such as partial failure, imperfect failure detection and non-transitive connectivity. We will discuss these three concepts more in detail later in this chapter.

Coming back to basics of distributed programming, let us review two definitions that target some key concepts concerning distribution. According to Tanenbaum and van Steen [TV01]:

> “A distributed system is a collection of independent computers that appears to its users as a single coherent system”

This definition suggests using *distribution transparency*, where all the effort of distributed programming is moved to the construction of a middleware that supports the distribution of the programming language entities. But network and computer failures cause unexpected errors to appear at higher abstraction levels, which breaks transparency and complicates programming.

The key issue in distributed programming is partial failure. It is what makes distributed programming different from local concurrent programming. In concurrent computing, programs are designed as a set of tasks that could run in parallel. We call it local concurrency when all these tasks run on a single processor. Communication between different tasks is always guaranteed and tasks do not fail independently. When tasks are executed in different processors located in different machines, tasks really run in parallel, but the communication between them cannot be guaranteed all the time, and tasks can fail independently. A partial failure occurs when one of processes running a task crashes unexpectedly or when the communication link between two processes is broken. This is why we would like to quote Leslie Lamport and his definition of a distributed system:

> “A distributed system is one in which the failure of a computer you did not even know it existed can render your own computer unusable”

It does not matter how much transparency can be provided in distributed programming, it will always be broken by partial failure. This is not particularly bad, but it means that we need to take failures very seriously. We must consider that a failure does not mean only the crash of a process, but also a broken link of communication between two processes. This condition implies two important consequences: connectivity between processes is not transitive, and perfect failure detection is mostly unfeasibly, specially on Internet style networks. This is because it is impossible to distinguish between a process that stop working from a broken or slow communication link. Perfect failure detection means that any crashed process will be eventually detected by all alive processes, and that no alive process will ever be suspected of having failed. On the Internet, and in many other networks, the best that can be achieved is *eventually perfect* failure detection. That means that all crashed processes will be eventually detected by all alive processes, and that any false suspicion of an alive process will be eventually fixed. This imply another relaxation to cope with reality: do not request the system to provide perfect failure detection, but rely only on eventually perfect failure detection.

Assuming that any node can directly communicate with any other node is surprisingly another strong requirement that it is often broken. If a process a can talk to a process b, and process b can talk to process c, it does not imply that a can talk to c. This is what is called *non-transitive connectivity*. An example of how this situation can arrive is the following. Assume processes a is running behind a network address translation (NAT) device, as many of the routers provided by home Internet providers. Process b is running with a global IP address, so that a can talk to b directly. Process c is behind another NAT device and b can also talk to c. However, if these NAT devices are not configured to allow communication between peers without a public IP address, a and c will not be able to establish a direct communication between them. Currently, there are techniques called NAT traversal to get around this problem, but there is no guarantee of working in all cases. Therefore, relying on transitive connectivity is still a strong requirement on the system that needs to be relaxed.

Resuming our discussion on partial failures, we should not trust the stability of the whole system to a single node, or to a reduced set of nodes with some hierarchy. We need to build self-managing decentralized systems, where data storage needs to be replicated and load balanced across the network in such a way as to provide fault tolerance.

One of the results presented in this dissertation is the *relaxed ring*, a structured overlay network topology which is fully decentralized, self-organizing, self-healing, scalable and that deals with non-transitive connectivity relying only on eventually perfect failure detection. Another important contribution of this work concerns robust storage in peer-to-peer systems. We provide support for a transactional distributed hash table with replication, allowing developers to write applications for synchronous and asynchronous collaborative system. We combined these results to build Beernet, a scalable þeer-to-þeer network. The word þeer mixes *beer* and *peer* because Beernet is both: a peer-to-peer network, and a beer-to-beer network. Beer is a known means to achieve relaxation, and therefore, it reflects the philosophy behind the design of Beernet.

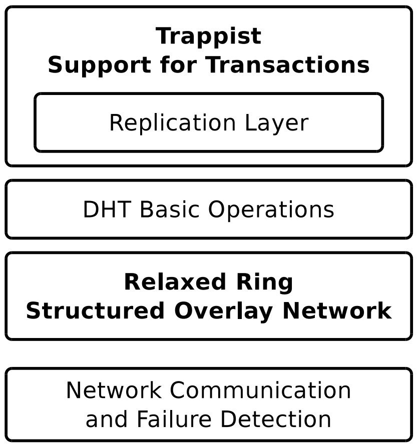


Figure 1.1: Global view of Beernet's architecture

We evaluate the algorithms and the implementation of the system, and describe a set of applications implemented with it.

A global view of Beernet's architecture is shown in Figure 1.1. The architecture is organized in layers preventing errors at lower layers to be propagated as failures at the upper layers. We will discuss more about this property in Chapter 7. The first part of this Introduction has been mostly dedicated to discuss the basic principles of distributed systems, which corresponds to the bottom layer in the architecture. Section 1.1 is about the main ideas behind the two middle layers, structured overlay networks and distributed hash tables (DHTs). What follows in Section 1.2 is an introduction to replicated storage, discussing the main challenges of it. Figure 1.1 highlights two of the contributions of this dissertation, the relaxed ring and Trappist, which is the layer providing transactional support to the replicated storage.

1.1 Scalable Systems

As we already mentioned, decentralized systems are gaining a lot of popularity in distributed systems thanks to scalability and fault-tolerance. While randomly connected networks are still used for file-sharing, hybrid architectures are preferred by commercial applications, specially when they want to build more complex systems that still needs the scalability provided by peer-to-peer technologies. In hybrid architecture, part of the system runs on a peer-to-peer network, and part is controlled by servers hosting a given service, for instance authentication. In the world of academia, structured overlay networks receive most of the attention, mainly because of their stronger guarantees and their self-managing properties. After ten years of research, this area seems to have achieved a very good theoretical understanding of how structured overlay networks should be designed. However, too few real systems have been built successfully, spotting that some of the basic assumptions were wrong, such as the transitive connectivity. Therefore, we believe, and we will show, that there is room for improvement in structured overlay networks, and that they will be the one of the bases to build scalable systems.

The programming concept that is strongly associated with structured overlay networks is the distributed hash table (DHT). A DHT is just like a hash table where the address space is divided among all participants of the network. The hashing function is known by all participant, and it is a local operation. What is distributed is the storage of table's data. The most used strategy for DHT is the circular address space having hash keys going from 0 to $N - 1$. Every peer is identified with a key so that they have total order between them, organized as successors and predecessors. Each peer is responsible for the storage of all data associated with hash keys in the range defined by its predecessor and itself, excluding the predecessor. The technical details of DHT will be explained in detail in Chapter 2.

The idea behind a DHT is strongly connected to the concept of *transactive memory* [WER91] in sociology, and we will use it to intuitively explain how a DHT works. The idea is that people do not have to remember every information they need. They just need to know *who-knows-what*. For instance, in a marriage, it is a common situation that only one of the partners knows everything about the finances of the house. If the other partner needs a particular information, let us say, how much do they spend every month in electricity, he just needs to ask the other partner. In a DHT it is the same. If a peer needs to know the value associated to a given key k, it just needs to ask the responsible peer of such key. Of course, it can happen that the peer is not directly connected to the responsible of k. In such case, its request will have to be routed through its direct neighbours, and then through the neighbours of the neighbours until reaching the responsible. The same thing happen in transactive memory, which relies on social networking, *friend-of-a-friend*, to find the person who knows the needed information.

Taking the example of the social network, we can understand why structured overlay networks are much more efficient than unstructured ones. Social networks form a small network graph, sometimes called *Human Web*. Every person can reach any other person in a maximum amount of hops, which is the diameter of the network. This often referred as the *six degrees of separation* [Bar03], meaning that in theory one could build a chain of around six *friend-of-a-friend* links to reach any other person. The problem is to know which friends to consider in the chain. In other words, to find the path to reach the destination. Unstructured overlay networks use flooding routing to find the responsible of a given data. That would be as if one ask all its friends for a given information they do not have. Then, each one of the friends will ask all its own friends the same question propagating the search, until reaching the responsible. If the request can be answered immediately, this is unknown to the other participants. Therefore, the request will be propagated anyway flooding the network. In structured overlay networks, every peer knows exactly which peer is closer to the responsible of the key, and therefore, it will route the request efficiently always making progress towards the responsible. If we come back to the transactive memory concept, we understand why companies have a well defined structure were responsibilities and hierarchies define *who-knows-what*, and it is possible to find the *who* in a more efficient way than simply flooding (at least in theory).

1.2 Replicated Storage

Following with the concept of transactive memory, a well-organized organization will have more than one person as responsible for any given piece of information. If one of the responsible persons is not able to continue being part of the organization, its information is not lost because there is always another person who also knows the information. That is one of the main goals of replication: fault tolerance. Another goal is quick access to the information. The situation is also valid for DHTs. If more that one peer can answer the query of a given key, the network can become more fault-tolerant. Of course, if the two replicas leave the system simultaneously, the information will be lost any way. Therefore, the amount of replicas define the tolerance factor of the systems.

Replication is not for free. It is necessary to synchronize the replicas to avoid inconsistencies in the retrieval of information, and also to prevent data lost. The more replicas there are, the more costly is the synchronization among them, but the higher is the fault-tolerance factor. The fault-tolerant factor defines the amount of failures that the system can tolerate to keep on working and still provide a service. Every system must decide its optimal amount of replicas according to higher-level policies. The synchronization of replicas must guarantee that when a replica is asked for the value associated to a given key, it answers consistently with any other replica holding the same key. Another possibility is that the system guarantees that by asking to the majority of the replicas it is always possible to deduce the last up-to-date value. The update of replicas must also take into account that new participants can join the network while a synchronization is running and, even more difficult to handle, some of the replicas can suddenly leave the network while an update is taking place.

1.3 Self Managing Systems

We have intuitively described what a DHT is, and motivated why structured overlay networks are more efficient in finding information than unstructured networks. We have also motivated replicated storage, briefly explaining the challenges implied by the maintenance of synchronized replicas. All these arguments will be analysed properly and discussed in the following chapters of this dissertation. We also mentioned the need for self-management to deal with the complexity of decentralized system, given the lack of a single point of control. A self-managing system has the ability to heal itself when some of its parts fail. Its parts are also able to find a working configuration without needing the intervention of an external agent. They are able to find such a configuration just by studying their internal state and comparing it to the one of their local neighbours. In the same way, these kind of systems are able to self-organize and provide other self-* properties, such as self-tuning, self-adaptability and self-protection.

As explained in several works on self-managing systems [Van06, KM07, Van08, Bul09, KC03], a fundamental property that such a system must have is to constantly monitor itself, analyse the collected information, decide an action to modify the system if needed, perform the action, and monitor again to continue with the process. Each of these tasks can be done by a dedicated independent component or by a set of them. It is essential that the system runs in a permanent *feedback loop* to be able to react to any perturbation that breaks its stability. The concept of self-management is actually very close to self-stabilization. We will discuss self-* properties across all the chapters of this dissertation, dedicating Chapter 4 to the analysis of the algorithms of our structured overlay network using feedback loops.

1.4 Thesis and Contribution

The thesis of this dissertation proposes a design philosophy: the relaxed approach. The contributions of this work are the result of applying this philosophy to the design and construction of distributed system aiming scalability and robust storage.

> The relaxed approach is a design process for systematically relaxing system requirements to allow it to cope with the inherent asynchrony of distributed systems without sacrificing functionality. We successfully apply this approach to build scalable distributed systems with self-managing behaviour and transactional robust storage.

The thesis is supported by the following contributions:

- By relaxing the ring structure for peer-to-peer networks, we were able to cope with non-transitive networks and false suspicions in failure detection. The result is the relaxed ring topology, a self-organizing and self-healing structure for peer-to-peer networks that improves lookup consistency and reduces the cost for ring maintenance. The relaxation introduces branches to the ring topology, but it keeps the routing algorithm competitive with $O(log_k N)$ hops to reach any peer.
- By relaxing the logarithmic construction of the finger table, we provide a self-adaptable routing topology that allows the relaxed ring to take

advantage of full connectivity in small networks, and logarithmic routing in large networks. The system can scale up and down making it suitable for many different applications independent of the size of the network.

- Paxos consensus algorithm relaxes the requirements on replicated storage relying only on the agreement of the majority of replicas instead of on all of them. We adopt and adapt Paxos to provide eager locking of replicas easing the development of synchronous collaborative applications. We extend both protocols with a notification layer to make other peers aware of replicas' updates.
- By relaxing versioning and ordering of elements in data collections, we designed a lock-free transactional protocol for key/value-sets. The protocol outperforms key/value pairs to manipulate unordered set of values. We guarantee strong consistency when data is read from the majority of replicas.
- We complement the relaxed approach with feedback loops, a design tool for self-managing systems. We use feedback loops to identify patterns of self-managing behaviour in the relaxed ring, and we use them to design the above mentioned self-adaptable routing table, and a self-tunable eventually perfect failure detector.

What follows is a list of work done to achieve the above mentioned contributions:

- We studied and validated the Paxos consensus algorithm for atomic transactions on a replicated DHT, and we compared it with the well known solution for distributed transactions called Two-phase commit.
- We have implemented Beernet, including the relaxed ring and Trappist. Peers are organized as a set of distributed-transparent actors. These actors represent components with encapsulated state and that communicate only via message passing, avoiding shared state concurrency. Beernet also takes advantage of the fault-stream model for failure handling improving its modularity and network transparency. These characteristics provide a better programming framework for self-configuring components.
- We have implemented and presented to the research community three different demonstrators to introduce the concepts of the relaxed ring, atomic transactional DHT, and synchronous collaboration with eager-locking transactions.
- We developed two applications on top of Beernet to exploit optimistic and pessimistic transactions, and the notification layer. These applications provide a community-driven recommendation system, and a collaborative drawing tool. A third application designed and developed by third parties is also presented to emphasize the impact of the contribution of the relaxed ring and Trappist.

1.5 Publications and Software

The work presented in this dissertation is the result of incremental progress made through discussions and presentations in several conferences, workshops and doctoral symposia. The implementation of software as proof-of-concept has contributed to polish algorithms and ideas about how decentralized systems should be designed. Several of the results presented here have also been published in conferences and journals. In this section we present the most important publications that support this dissertation, together with references to software demonstrators that validate the implementation of the ideas. There is also an award to be mentioned that the author received for his presentation in a doctoral symposium.

- **Journal** *"Beernet: Building Self-Managing Decentralized Systems with Replicated Transactional Storage".* Boris Mejías and Peter Van Roy. In IJARAS: International Journal of Adaptive, Resilient, and Autonomic Systems, Vol. 1(3):1-24, IGI Publishing, July 2010. This paper briefly summarizes the contribution of the relaxed ring to focus on the design and implementation of Beernet and Trappist, which are presented in Chapters 7 and 5 respectively. It also describes the design of the applications built on top of Beernet, which are described in Chapter 8.

- **Journal** *"The Relaxed Ring: A fault-tolerant topology for structured overlay networks".* Boris Mejías and Peter Van Roy. In Parallel Processing Letters, Vol. 18(3):411–432, World Scientific, September 2008. This publication presents most of the results that we will describe in detail in Chapter 3, presenting part of the evaluation measurements that will be presented in Chapter 6.

- **Conference** *"A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks".* Boris Mejías and Peter Van Roy. In Proceedings of XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 8-9 November 2007, Iquique, Chile. This publication is focused on the analysis of the self-management behaviour of the relaxed ring, using feedback loops as a mean to describe an analyse the algorithms for ring maintenance and failure recovery. These ideas are further discussed in Chapter 4.

- **Conference** *"PALTA: Peer-to-peer AdaptabLe Topology for Ambient intelligence".* Alfredo Cádiz, Boris Mejías, Jorge Vallejos, Kim Mens, Peter Van Roy, Wolfgang de Meuter. In Proceedings of XXVII IEEE International Conference of the Chilean Computer Science Society (SCCC'08). November 13-14, 2008, Punta Arenas, Chile. This paper complements relaxed ring's algorithms with an efficient self-adaptable routing table. The algorithm is described in detail in Chapter 3, and its evaluation is included in Chapter 6.

- **Demonstrator in Conference** *"PEPINO: PEer-to-Peer network INspectOr"* (abstract for demonstrator). Donatien Grolaux, Boris Mejías, and Peter Van Roy. In Proceedings of P2P '07: Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing. September 2-7, 2007, Galway, Ireland. This software demonstrator has helped us to visualize and polish our implementation of the relaxed ring, being closely related to Chapter 7 and the applications of Chapter 8.
- **Demonstrator in Conference** *"Visualizing Transactional Algorithms for DHT"* (abstract for demonstrator). Boris Mejías, Mikael Högqvist and Peter Van Roy. In Proceedings of P2P '08: Proceedings of the Eighth IEEE International Conference on Peer-to-Peer Computing. September 8-11, 2008, Aachen, Germany. This software demonstrator validates the design and implementation of the transactional layer for atomic commit on DHTs with symmetric replication. It validates the claims we discuss in Chapter 5.
- **Software Released** Beernet 0.7 was released under the Beerware free software license in April 2010. Beernet implements the relaxed ring and the Trappist transactional layer. Previous to the last release, a Lightning Talk was presented at FOSDEM'2010 in February. FOSDEM is one of the most important European meetings in free and open source software.
- **Award** The author has won the *"Best Presentation Award"* at the Doctoral Symposium of the "XtreemOS Summer School", held at the Wadham College of the University of Oxford, Oxford, UK, on September 10, 2009. The presentation was entitled "Beernet: a relaxed ring approach for peer-to-peer networks with transactional replicated DHT" [Mej09], and it summarized the contribution of this dissertation.

1.6 Roadmap

This dissertation is organized as follows. Chapter 2 makes a review of all three generations of peer-to-peer systems, where structured overlay networks are the main focus of the analysis. The systems we reviewed are not only studied from the point of view of their overlay graph, but also from their self-managing properties. We also review distributed storage and the connection of peer-to-peer with Grid and Cloud Computing. Figure 1.2 helps us to describe the organization of the rest of the dissertation. Chapter 3 presents the protocols and algorithms of the relaxed ring, being an important part of the contribution of this dissertation. The relaxed ring is also studied using feedback loops in Chapter 4 to understand its self-managing properties from an architectural and software design point of view. Feedback loops complement the relaxed approach as design tool for self-managing systems.

Once we have presented the relaxed ring, the dissertation continues in Chapter 5 with the description of Trappist, our contribution in distributed storage.

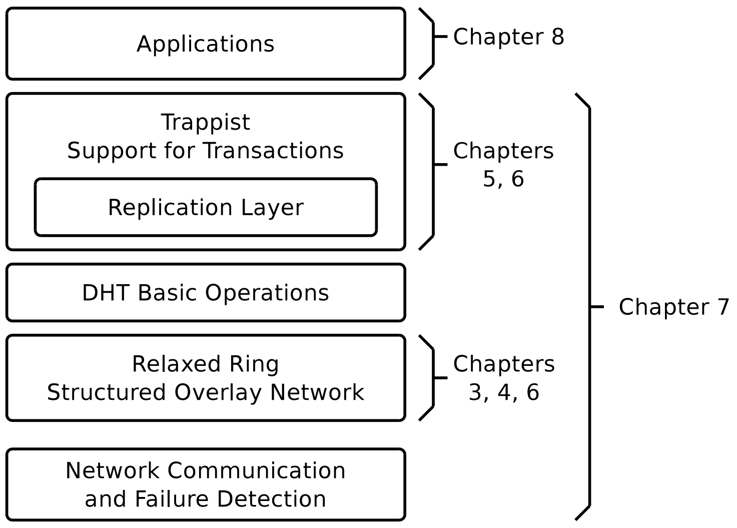


Figure 1.2: Dissertation's roadmap.

We analyse Two-Phase commit, Paxos consensus algorithm, and we describe our contribution to Paxos extending it with eager locking. We also present our notification layer, and the lock-free protocol for key/value-sets, which is our support for data collections. Discussions related to DHT are covered in Chapters 3 and 5. Evaluation of relaxed ring's algorithms in comparison with other overlay networks is presented based on empirical results in Chapter 6. We also include an analysis on the impact of network address translation (NAT) on ring-based overlay networks. A performance analysis of the different transactional protocols provided on this dissertation is also included in this chapter. Therefore, Chapter 6 contributes to validate the relaxed ring and Trappist. Chapter 7 describes the design decisions and implementation details of Beernet. In that chapter we provide more details about the advantages of its architecture.

In Figure 1.2, we have extended the architecture by adding the top layer representing the applications that are built on top of Beernet. In Chapter 8 we present three applications covering synchronous and asynchronous collaborative systems, and we also include an application developed by programmers other than the author. The dissertation finishes with concluding remarks and future work in Chapter 9.

Chapter 2

The Road to Peer-to-Peer Networks

All animals are equal, but some are more equal than others

"Animal Farm" - George Orwell

The goal of distributed computing is to achieve the collaboration of a set of different autonomous processes. A process is an abstraction of an entity that can perform computations. This entity can be a computer, a processor in a computer, or a thread of execution in a processor. We will use *nodes* or *peers* to also mean a process. The most basic problem that has to be addressed is to establish the connection between two processes and to provide programming language abstractions to allow programmers to perform distributed operations. As more processes come into communication, enlarging the network, it is necessary to correctly route messages between processes that are not directly connected. And as the network grows larger, it is necessary to design system architectures that can ease the collaboration between processes. Even though the first issues we mentioned are not completely solved, the existing solutions has been successfully used to communicated distributed processes, so they are good enough to let us focus on the architecture of the system.

We are interested in designing and building overlay networks to organize processes. An overlay network is a network built on top of another network, which it is call the underlying network. The underlying network provides communication between the processes we want to organize on the overlay network. This chapter is dedicated to analyze existing overlay networks to contextualize the contribution of this dissertation. Even though this work is developed at a high level of abstraction, we still consider many of the basic principles of distributed computing, such as *latency* or *partial failure.* These principles go across all levels of abstraction of distributing computing, and not taking them into account would be like an architect discarding physical rules that would invalidate a design of a building. We will mention some issues concerning routing messages on the underlying network during this chapter, and we will discuss more about language abstractions for programming languages when we describe the implementation of Beernet in Chapter 7.

In the introduction chapter we motivated the use of peer-to-peer networks for building dynamic distributed systems, because of their decentralized, fault-tolerant and self-organizing structure. We claimed that by relaxing the strong requirements, we make the system more realistic without sacrificing functionality. We also claimed that increasing self management in such systems is the only way of dealing with their high complexity. The self-management properties of peer-to-peer networks are so intrinsic to them that we will start this chapter by briefly introducing some concepts of self management, and then we will use them to analyze the related work.

2.1 Self Management

The complexity of almost any system is proportional to its size. This rule also holds for distributed systems. As systems grow larger, they become more and more difficult to manage. Therefore, increasing systems' self-management capacity appears as a natural way of dealing with high level complexity. By self management, we mean the ability of a system to modify itself to handle changes in its internal state or its environment without human intervention but according to high-level management policies. This means that human intervention is lifted up to the level where policies are defined.

Typical self-management operations are: tune performance, reconfigure, replicate data, detect failures and recover from them, detect intrusion and attacks, add or remove parts of the system, which can be components within a process, or a whole peer, and others. Each of those actions or a combination of them can be identified as *self-configuration, self-organization, self-healing, self-tuning, self-protection* and *self-optimization*, often called in literature *self-** properties. We will use these properties to analyze the related work and the contribution of this work, but self-protection is not in the scope of this dissertation.

One of the key operations that a system must perform to achieve self-managing behaviour is to monitor itself and its environment. Once relevant information is collected, the system can take decisions over which actions to trigger to achieve its goal. Once an action is triggered, the system needs to monitor again to observe the effect of its action, developing a constant feedback loop. We will explain feedback loops in Chapter 4. In peer-to-peer systems, monitoring is distributed and based only on the local knowledge that every peer has. Peers monitor each other and trigger actions in other peers. Global state can be inferred but always as an approximation, because there is no

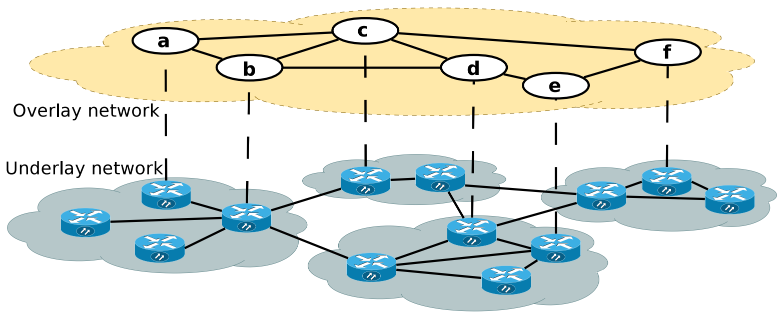


Figure 2.1: Overlay network.

central point of control that observes the whole system at once. Self-managing behaviour must be observed as a property of the whole network, and not as an isolated property of a single peer.

2.2 Overlay Networks

A computer network is a group of interconnected processes able to route messages between them. Internet is a group of interconnected networks, routing messages between processes independently of the network where they belong. An *overlay network* is a network built on top of another network or set of networks. For instance, a group of processes using the Internet to route their message is said to be an overlay network, where the Internet is the *underlay network*. Actually, the Internet itself can be seen as an overlay network running on top of the group of local area networks.

Figure 2.1 depicts the architecture we are describing. An important issue to discuss here is the analysis of the routing of messages. We can observe that nodes **f** and **e** are directly connected in the overlay network. Therefore, a message sent from node **f** to **e** is considered to be sent in one hop. However, if we look at the topology of the underlying network, we observe that the message would take at least three hops, and it goes through the node identified as **d**, which is not even connected with **f** in the overlay. This difference in the amount of hops is understandable if we consider that the overlay network completely abstracts the underlying network. The same overlay network depicted in Figure 2.1 could have been deployed over a different underlying network where nodes **f** and **e** are really directly connected, or completely far away. Since there is no direct correlation between overlay and underlay in the amount of hops needed to route a message, we will consider these two analyses as independent. This does not mean that the design of an overlay network cannot take the underlying network into account to optimize routing. In this dissertation, when we discuss the amount of hops needed to route a message we mean hops at the level of the overlay network, unless it is explicitly stated otherwise.

2.2.1 Client-Server Architecture

Client-Server is the one of the most basic and popular architectures to build distributed systems. It is very simple and it allows the designer of the server to have control over the system, because all messages have the server as participant. It can also be seen as an centralized overlay network with a *star* topology. Unfortunately, it relies too much on the server which becomes a point of congestion and a single point of failure. The system relies entirely on the server. If the server crashes, there is no application. The size of the application also relies on how powerful the server is to handle the connection of all the clients. Therefore, it does not scale. If there is any self-management property that we want to analyze here, it would be entirely focused on the server, and we would remain with the problem that if the server is gone all self-* properties will be gone too.

Currently, companies that base their business model on the client-server architecture have extended it to run more code on client's machine, allow some communication directly between clients, and more fundamentally, replicate their servers to scale and provide fault tolerance. If we focus the self-management analysis on the group of servers, then we would not be studying a client-server architecture anymore, because the group of servers would actually form a different network. Yet, if the access to the group of servers is broken, there is no application. To achieve more fault tolerance it is necessary to decentralize the system. Decentralization will also increase scalability but at the cost that there will be no central point of control. Increasing self management will allow the system to control itself.

2.2.2 Peer-to-Peer First Generation

Napster [Nap99] is the first peer-to-peer system to be widely known. It was a file-sharing service that allowed users to exchange files directly, without sending them though a server. It is said to belong to the first generation of peer-to-peer networks where we also find AudioGalaxy [Aud01] and OpenNap [Ope01]. This generation was not entirely peer-to-peer. It was based on a mixed architecture which still relied on a server to work. Peers connected to a server to run queries over media files. The server replied with the addresses of the peers storing the requested file so that peers could connect directly. If the server failed, the exchange of files could continue, but it was not possible to run new queries. Therefore, the application would stop to work as soon as all current downloads where completed. It was not possible to route messages to other peers through the currently connected peers, and this is why it is not considered to be entirely peer-to-peer. Only the exchange of files was done peer-to-peer. OpenNap, an open source derivate work from Napster, allowed communication between different servers improving robustness of the system, but the network remained centralized on the group of servers handling the queries. Because large part of the exchanged content was copyrighted, Napster ceased its operations in 2001 due to legal issues. The service was easily shut by just stopping the servers.

2.2.3 Peer-to-Peer Second Generation

The more it stays the same, the less it changes!

"The Majesty of Rock" - Spiñal Tap

The second generation of peer-to-peer networks is considered to be the first real peer-to-peer system, because it is fully decentralized, it does not rely on any server, and it is able to route messages using peers on the overlay network independently of the underlay network. This generation is mainly represented by Gnutella [Gnu03] and Freenet [Fre03], and it is also developed having file-sharing as goal. It was actually a solution to Napster's shut down, because there was no server to stop. These systems are also known as *unstructured overlay networks* because peers are randomly connected without any particularly defined structure. As we have discussed already, nowadays almost any machine can behave as a client and a server. Therefore, every peer can trigger queries as a client, and handle queries from other peers, playing the role of a server.

The algorithm to route messages in such unstructured network is called *flooding*. It is very simple but highly bandwidth consuming. It works as follows: the peer that triggers the query sends it to all its neighbours with a time to live (TTL) value. The TTL can be expressed in seconds or hops. We will use hops for our example. The receiver of the query determines if it is the first time that has seen it and if the TTL is greater than 0. If so, it transmits the query to all its neighbours except for the sender, with a decremented TTL. If the peer can resolve the query, it answers back following the path to the original sender. In Figure 2.2 we can see the flooding algorithm initiated by peer **h** with a TTL of 2. The extra circles around the peers on the figure represent the amount of hops that the message needed to reach the peer. All coloured peers participated in the routing algorithm. Peers with a darker coloured means that the peer receive the message more than once. Let us imagine that the query triggered by peer **h** can be answered by node **m**. The query is first sent from **h** to nodes **f**, **g**, **j** and **k**. Every peer will send it to its neighbours, so **m** receives the message from **k**, and the answer travels back following **m**→**k**→**h**.

There are several issues that make this algorithm less scalable and not suitable for the kind of systems we want to build. If we observe peer **f** on this example, first, it receives the query from peer **h** and then from peer **j**. A similar situation occurs with the other nodes on the first level of flooding. On the second level, nodes **i** and **n** receive the message twice too. In conclusion, many nodes process the query unnecessarily more than once, inefficiently using their resources.

A second issue is the amount of messages being sent. In this example, the messages is sent 16 times without counting the responses. In the ideal case,

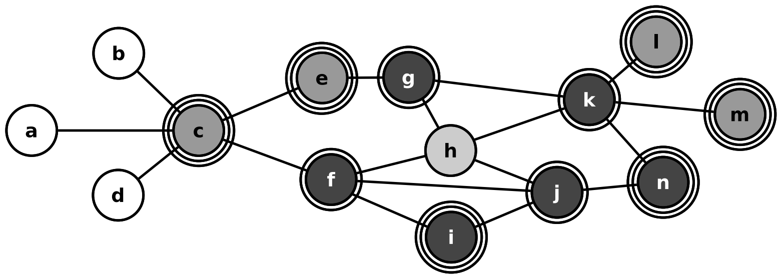


Figure 2.2: Flooding routing in a unstructured overlay network.

it was only necessary to follow the path **h**→**k**→**m**. To do such routing, peers would need a routing table with information about their neighbours. However, the absence of routing tables for the sake of simplicity is considered to be one of the advantages of unstructured overlay networks.

Determining a correct TTL is also an issue. If the query could have been resolved immediately on the first level, all the messages sent to the second level and further would have been unnecessary. If the TTL on this example would have been set to 3, the whole network would have been flooded to resolve the query. However, if the query could have been only resolved by **a**, **b** or **d**, a TTL of 2 would not have been sufficient to find the answer. This is one of the reasons why unpopular items are more difficult to find in file-sharing services based on flooding routing, even though the items are stored somewhere in the network.

Another issue that influences the success of resolving a query is the place of the originator. Nodes closer to the center of the network will reach more nodes that nodes living at the border of the network. For instance, in Figure 2.2, node **h** floods the whole network in three hops, but nodes **l** and **m** would need to use a value TTL of 5 to reach **a**, **b** or **d**. In a very large network, a TTL of 6 seems to be reasonable following the *six degrees of separation* theory of human connected networks [Bar03]. Using a formula from [AH02] to count the amount of messages sent in a query (factor 2 includes the responses), with an average of C connections per peer, using TTL = 6, and $C = 5$, we obtain 54610 messages per query:

$$2 * \sum_{i=0}^{TTL} C * (C-1)^i = 54610 \tag{2.1}$$

Despite all these issues, unstructured overlay networks are still very popular to run file-sharing services because bandwidth consumption is mainly considered a problem for ISP providers and not for the users, and because popular items can be found in a reasonable amount of hops, again, following the rule of *six degrees of separation*. Another reason is that items stored in file-sharing systems do not update their values. If many peers store the same item, it is necessary to find at least one of the peers. There is no such thing as the *latest value* of the item, so there is no issue with respect to consistency. Any replica of an item is a valid one as long as the value stays the same.

Flooding routing works fine for networks with a tree topology, because it avoids that peers receives messages more than once, but it is very costly for unstructured overlay networks, being inefficient in bandwidth and processing-power usage. Another problem is that there is no guarantee of reachability or consistency properties, which we consider important to build decentralized systems that constantly update the values of the stored items. Even though Gnutella can keep large amounts of peers connected, it does not mean that scalable services can be built on top of it because of the problems with efficiency, reachability and consistency [Mar02, RFI02]. Also, according to [DGM02], it is not difficult to perform a query-flood DoS attack in Gnutella-like networks, but their success depend on the topology of the network and the place of the originator of the attack, which is related to the reachability issue we already discussed.

Freenet also uses flooding routing but it presents some improvements with respect to Gnutella. The main difference is that the queries can be done with anonymity in Freenet. Since these systems were conceived as file-sharing services, the motivation for providing anonymity is basically legal, so no user can be sued. There is a degradation in performance because everything is sent encrypted. Freenet also keeps some information with respect to locality on the routing tables of the peers to improve routing speed. But, since it is flooding based, it is still very expensive.

With respect to self-* properties, we observe a basic self-organization mechanism despite the fact of not having any structured topology. There is no global or manual mechanism to organize the peers. Peers joining the network are just connected to the peers it gets introduced by its entry point. When nodes disconnect from the network, the other nodes simply stop forwarding queries to them, and therefore, there is no need for a self-healing strategy. Routing protocols in Gnutella and Freenet are under continuous improvement by their communities, but we will not refer to them because they go beyond the simplicity of the basic unstructured network, and they do not solve the more fundamental problems already discussed. It is possible to provide some self-tuning of the TTL value, and some self-optimization in the flooding paths by adding more information to the routing tables of the peers, but we will study better choices on the structured overlay networks in the next section.

2.3 Structured Overlay Networks

The third generation, also known as *structured overlay networks* (SONs), is the result of academia's interest in peer-to-peer networks. It clearly aims to solve the problems of unstructured networks by providing efficient routing, guaranteeing reachability and consistent retrieval of information. Adding structure makes it possible to achieve these improvements, but it also creates new challenges such as dealing with disconnections of peers and non-transitive connectivity. Non-transitive connectivity means that if a peer A can talk to B, and peer B can talk to C, it does not mean that peer A can talk to C. As we will see in this section, most approaches assume full transitive connectivity, which is not true for Internet-like scenarios. Many nodes connected to the Internet run behind network address translation (NAT) devices. These devices filter a lot traffic between nodes, and sometimes, they prevent direct communication between two NATted nodes, introducing non-transitive connectivity. Broken routing paths between nodes is also a source of non-transitivity, but those cases are more difficult to detect because they are mostly temporary. However, their influence in connectivity quality is not negligible.

Every system assumes that nodes willing to join the network have a reference to at least one peer in the system, which we call the first contact. That implies that networks should provide a way of publishing references to peers. SONs typically provide a distributed hash table (DHT) where every peer is responsible for a part of the hash table. There are two basic operations that every DHT must provide: `put(key, value)` and `get(key)`. The `put` operation stores the value associated with its key such that every peer can retrieve it with the `get` operator. If another value was already stored under the same key, the value is overwritten. We define now some of the concepts we will use in this section and in the following chapters. These terms are also common to other surveys found in the literature [AH02, BL03, GGG+03, EAH05, LCP+05, AAG+05].

- **Item.** The *key/value* pair stored in the hash table.
- **Identifier space.** The key of an item is always mapped into a hash key, which is the *identifier* of the item (abbreviated as id). The range of possible values of ids is the *identifier space.* Peers in the network are also associated with an id. Because of that, the identifier space is also named *address space*
- **Lookup.** It is the operation performed by any peer to find the responsible peer of a given key.
- **Join.** A new peer getting into the network.
- **Leave.** A peer disconnects from the network either voluntarily (gentle leave) or because of a *failure* (also named *crash*).
- **Churn.** Measurement of peers joining and leaving the network during a given amount of time.
- **Iterative routing.** The originator of a lookup sends its message to its best contact on the overlay, with respect to the searched key. The contact answers back with its best contact, so the originator iterates until reaching the destination. This seems to be inefficient, but if a node

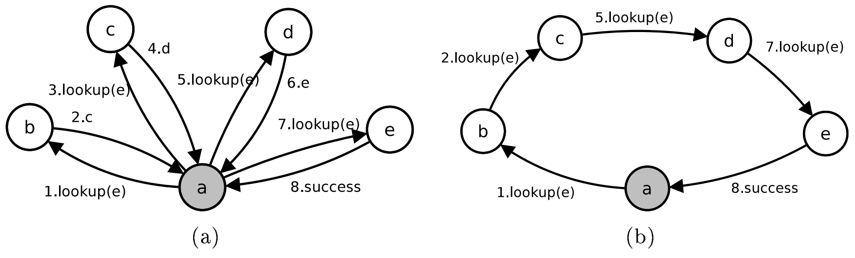


Figure 2.3: Example of (a) iterative and (b) recursive routing.

in the path fails, the originator knows exactly where to recover from. See Figure 2.3(a).

- **Recursive routing.** The originator of a lookup sends its message to its best contact on the overlay, with respect to the searched key. The contact forwards the lookup request to its best contact, and so continues the search. When destination is reached, the peer answers back to the originator. It is more efficient than iterative routing, but in case of a failure, the whole process needs to be restarted. See Figure 2.3(b).

We will start by discussing ring-based networks because it is where the contribution of the relaxed ring is made. Then, we will summarize other kinds of SONs to complete the review, explaining what are the advantages of ring-based structure, and how our global contribution could be applied to other networks. We will deeply analyze Chord because it influences many other systems, which are basically improvements of Chord.

2.3.1 Chord

Chord [SMK+01, DBK+01] is one of the most known and referenced SONs. In Chord, peers are self-organized forming a ring with a circular address space of size N. Hash keys are integers from 0 to $N-1$. The ring can be seen as a double-linked list with every peer having two basic pointers: *predecessor* and *successor* (abbreviated as *pred* and *succ*). Figure 2.4(a) depicts an example of a Chord ring. Only pointers of peer identified with id q are drawn on the figure but every peer holds equivalent pointers. Peers p and s corresponds to *pred* and *succ* respectively. This means that $p < q < s$, where '$<$' is defined on the circular address space following the ring clockwise.

DHT The ring provides a DHT where every peer is responsible for the storage of a set of keys determined by its own id and its predecessor. In the case of q, the peer is responsible for the range $]p, q]$, (i.e., excluding pred's id and including its own). If the ring is perfectly linked, there is no overlapping of peers' responsibilities, and therefore, every lookup operation gives consistent results.

Fingers To provide efficient routing, Chord uses a set of extra pointers called *fingers* or *long references*. They are chosen dividing the address space in halves. The farthest finger of q is the responsible of key $(q + N/2)$ mod N. In our example in Figure 2.4(a), we consider $q = 0$ to label finger keys, and therefore, the ideal farthest finger key is $N/2$. In the figure there is no peer holding exactly that key, but peer k is currently the responsible. Closer fingers are chosen using the same formula but dividing N by powers of 2. The fingers, together with pointers to pred and succ, form the routing table of a peer. Ideally, every peer holds references to $log_2 N$ fingers.

Lookup When any peer receives a lookup request for a given key, it first determines if the key belongs to the range between its own id and its successor. If that is the case, it answers the lookup query giving its successor id as answer. If it is not, it forwards the lookup to the closest preceding finger. The routing mechanism is therefore recursive. In our example of Figure 2.4(a), if `lookup(m)` is sent to peer q, q forwards it to peer k. Then, peer k will continue forwarding using its own routing table. If `lookup(b')` is sent to peer q, with $b < b' < c$, then q forwards the message to b, being the closest preceding finger of b'. Then, b answers that c is the responsible, because $b' \in]b, c]$. The fact that b answers that its successor c is the responsible of b' will be the source of inconsistencies under special cases of churn and connectivity problems, as it is described in [Gho06, MV08, SMS+08]. This occurs basically because b could not be aware of new node between c and the key b'. We will come back to this issue in Chapter 3.

Peers never use pred pointer to route a messages, even if pred is closer to the target than any finger. This rule is taken for efficiency and to prevent cycles. If every peer continues routing the message backwards, there is no guarantee that it would take less than $O(log_2 N)$ hops to reach the responsible. Furthermore, is one of the peers decides to switch again from backward routing to forward routing. It could reach a peer which would backward the message again, creating a cycle. Therefore, only one direction is used for the routing in Chord.

Fingers fragment the address space into halves, therefore, every forwarding of a lookup request shortens the distance to approximately the half of it. Considering that the address space is discrete, the routing of the lookup converges to the responsible of the key in $O(log_2 N)$ hops. This routing cost is very scalable because if the network doubles its size, the routing takes in average only one extra hop.

Churn Figure 2.4(b) shows three different events producing churn: peer j joins as k's predecessor, peer b leaves voluntarily the network, and peer m

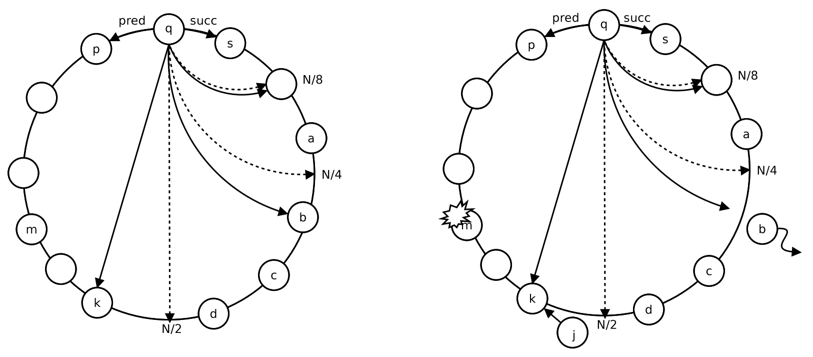


Figure 2.4: Example of (a) Chord ring and (b) some events causing churn.

crashes. In the case of the join, k accepts j only if j belongs to k's range of responsibility. Since $N/2 > j > q$, peer j becomes the new responsible of $N/2$, and therefore, it is a more suitable finger for q. This value needs to be updated somehow. A similar situation occurs when b leaves, because c becomes the new responsible of $N/4$. The difference here is that now q has temporarily no finger for that value until it knows about c. The crash of m does not affect q's routing table, but it surely affects other peers' routing table, and the responsibility of m's successor.

Join When new peers want to join the network, they have to do it as predecessor of the responsible of their own key. Looking at the example in Figure 2.4(b), we observe that peer j joins as predecessor of peer k. To know where to join the ring, j has to previously request a `lookup(j)` to its first contact, which can be any peer in the network. Since the answer to the lookup is k, peer j set its succ pointer to k and notifies it. Then, k determines that $j \in]d, k]$ and update its pred pointer to j. Half of the process of joining is done here. It is continued by periodic stabilization, which we describe now.

Periodic stabilization We can divide a peer's routing table into two groups: fingers, which are used for efficient routing, and pred/succ pointers, which are needed for dividing the address space. These references become invalid after some time due to churn. Therefore, it is necessary that every peer periodically checks the validity of pred/succ pointers. Analysing this strategy from the point of view of our relaxed approach, we can see that the Chord's join process is *too* relaxed. It just let nodes join the ring by talking to its successor, moving the larger load of work to periodic stabilization to achieve correctness. The problem actually arises because the loose join contradicts the strong requirement of having a perfect ring for correctness. The relaxation is somehow misplaced.

A peer periodically asks its succ for the value of succ's pred. If it is the same as itself, there is nothing to do. If it is a new one, it is probably a better successor, or something went wrong and there is an inconsistency in the responsibilities of the DHT. Coming back to our join example in Figure 2.4(b), when it is time for d to run periodic stabilization, it asks k the value of its predecessor. Peer k answers j. Peer d realizes that $j \in]d,k]$, and then, d changes its succ pointer to j, fixing the ring. Then, d notifies j about itself becoming j's pred. That is how j gets to know d and the joining of j is completed. This mechanism relies entirely on *network transitivity* (i.e., if d can talk to k and k can talk to $j \Rightarrow d$ can talk to j). This property is often assumed as guaranteed, but it does not hold all the time due to non-transitive connectivity, as it was explained in Section 2.3. This broken assumption is the source of errors in real implementations [FLRS05], creating lookup inconsistencies and uncorrected false suspicions. Another problem with this join algorithm based on periodic stabilization is that two peers joining simultaneously in the same range of keys will introduce lookup inconsistency even if connectivity is perfect, as analyzed in [Gho06]. We will discuss more about these two issues later on this section and in Chapter 3.

To check validity of fingers, a peer asks to every finger its pred value. If pred is the new responsible for the ideal finger key, the finger pointer is updated. In Figure 2.4(b), q asks k for its pred. Peer k answers j, where $j \in]N/2,k]$, and then, j is a better finger because it is the responsible for key $N/2$. Note that this mechanism also relies on transitive connectivity between q, k and j. Figure 2.4(b) also presents the example of peer b leaving the network. Peer b is the responsible of key $N/4$, which is one of the ideal fingers of q. When finger b is found to have left the ring, a new lookup for the key $N/4$ must be performed to find the new finger.

Successors list Every peer holds a list of peers that follows its successor clockwise, which we call successors list. The size of this list is $log_2 N$, as in the finger table. When the current successor leaves the network, either voluntarily or due to a crash, the peer takes the closest peer in the successors list as its successor candidate, fixing the ring. It is possible that the successors list is not accurate due to churn, and some peers will be missing, but this problem is corrected by periodic stabilization.

Network partitions Network partition is one of the worse scenarios of partial failure in distributed computing. It happens when two or more groups of nodes get disconnected from each other. An example of such situation is when the communication between two clusters is broken. The nodes within one cluster can communicate without problems, but they cannot contact any node in the other cluster, and it is impossible to know if the link is broken, or the cluster stop working. Chord can survive a network partition as long as every peer can find a valid successor candidate in its successor list. This means that no more than $log_2 N - 1$ consecutive peers have to reside on the same partition.

Even when the ring survives the partition, it is not possible to provide consistency and availability at the same time. This is not a particular problem of Chord but of every network following Brewer's conjecture on partition-tolerant web services, formalized and proven in [GL02] as the CAP theorem. Given the properties of strong consistency (C), high availability (A) and partition tolerance (P), the CAP theorem establishes that only two of the properties can be guaranteed sacrificing the third one. Even though Chord and other ring-based systems can survive network partitioning, none of these systems addresses correctly the problem of merging the rings when the partition is gone. Recently, a nice gossip-based solution was presented in [SGH07, SGH08], being general enough to apply it to many ring-based networks. Note that this dissertation does not address the issue of network partition. Beernet inherits the ability of surviving network partition and can integrate the above mentioned solution for merging rings.

Self management Note that no central entity organizes a peer's position in the ring. Every joining peer finds autonomously its successor, and every peer runs periodic stabilization independently. We identify this behaviour as *self-organization*, and it is essential to almost every SON. Periodic stabilization also reconfigures the finger table to provide efficient routing on the network. Considering fingers update only from the point of view of a single peer, we identify this behaviour as *self-configuration*. If we consider the global result, we observe that the network route messages more efficiently, therefore, we identify this behaviour as a basic *self-optimization*. The resilient information of the successors list, combined with periodic stabilization, can be clearly identified as *self-healing*.

It is important to remark that these self-management behaviours are intrinsic to Chord, as they are to almost every decentralized peer-to-peer network. Without these properties the system basically does not exist. This contradicts some views on self management that attempt to analyze decentralized systems as autonomic systems that have evolved from manually controlled systems. For instance, some methodologies [Mil05, LML05, ST05] define their model to evaluate the system *with* and *without* autonomic behaviour. They define the maturity of the system by the ability of turning on and off each autonomic behaviour. As we said, Chord would not work correctly without periodic stabilization, and one could not *turn off* self-organization of the ring. According to those methodologies, that would mean that the system is not mature enough, which actually does not reflect what a decentralized system is.

It would be more interesting to discuss how hidden a self-* property can be. For instance, the lookup procedure is orthogonal to the protocols that maintain the ring and the routing table. Therefore, the four self-* properties we already mentioned are hidden to the lookup. When the message arrives to the peer and it needs to be forwarded, the mechanism does not need to know how the pointers were defined. It just takes the most convenient finger. We realize that even when lookup is a low-level primitive in SONs, it is at a higher level with respect to routing table and ring maintenance.

Observations One of the advantages of Chord is that their protocols for maintaining the ring are quite simple and without locking the state of peers. We will discuss more about locks when we review DKS in Section 2.3.2. Unfortunately, as we already mentioned, the loose join strategy contrasts with the strict requirement of having a perfect ring to provide lookup consistency. Because they relax the join algorithm but not the requirements, they need to use a expensive periodic stabilization to maintain the perfect ring and to fix lookup inconsistencies. They also rely on transitive connectivity to complete the maintenance protocols. It is shown in [KA08] that there exists a given value for the ratio of churn with respect to the frequency of periodic stabilization, such that the longest finger of any peer is always dead at the moment of performing a lookup. This means that routing efficiency is highly degraded preventing the correct execution of any application built on top of the network. To solve this problem, periodic stabilization has to be triggered more often. We already mentioned that the join algorithm is not lookup-inconsistency free. Since periodic stabilization fixes those inconsistencies, making it run more often also contributes to a better ring maintenance. The big disadvantage is that periodic stabilization is very costly, making an inefficient use of the bandwidth.

The circular address space, as it is used by Chord and many other ring-based systems, relies on the uniformity of the identifiers of the peers. If the keys present a skewed distribution, many fingers will point to the same peer creating points of congestion. Another problem that can appear, even if peers' identifiers are uniformly distributed, is that the keys of the stored items have a skewed distribution. For instance, consider the words in a dictionary. If every peer has to store the words of a given letter, some peers will have to store a lot more information than others, unbalancing the network. This issue is addressed in Oscar [GDA06], a SON periodically reassign keys to achieve load balancing.

Chord was designed to scale to very large networks and it does it well, providing logarithmic routing cost. Unfortunately, if we would like to use Chord to create very small systems, the topology and routing tables would be too sophisticated and less efficient than a full mesh, which is completely reasonable to use in very small networks. Chord scales up very well, but it is not its goal to scale down. There are several Chord implementations and services built upon it, among which we find the main implementation [Cho04] and *i*3 [SAZ$^+$02], a DNS service [CMM02] and a cooperative file-sharing service [DKK$^+$01].

2.3.2 DKS

Overlay DKS [AEABH03] is also a ring-based peer-to-peer network with a circular address space like Chord. Its design presents improvements in routing

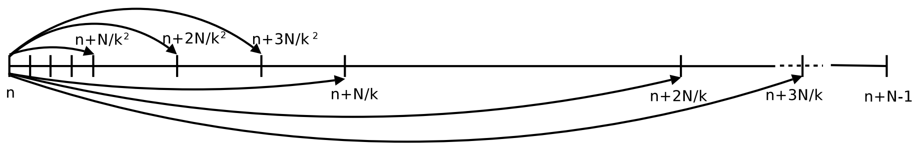


Figure 2.5: Partition of the address space in DKS.

complexity, cost maintenance of the ring, and replication of the data. DKS stands for Distributed k-ary Search. As it names suggests, the address space is divided recursively into k intervals rather than 2 as in Chord. An example of the division strategy can be seen in Figure 2.5. In the example, node n divides the space into $k = 4$ intervals, having a finger to each peer at the beginning of every interval. The closest interval is again divided into k subintervals with the correspondent fingers. The figure shows a new division of the closest interval, even though the arrows of the fingers are not drawn. The division continues until an interval is not dividable by k any more. The lookup process works exactly as in Chord for $k > 2$, forwarding the message to the closest preceding finger. Since there are always k intervals, the lookup process converge in $O(log_k N)$ hops, which is better than Chord. The larger the value of k, the smaller the amount of hops, but the larger the size of the routing table, which is a disadvantage because its maintenance becomes more costly. We can say that DKS generalizes Chord, where Chord becomes an instance of DKS where $k = 2$.

Correction-on-change and correction-on-use Another fundamental improvement with respect to Chord is that DKS does not rely on periodic stabilization. This can be achieved by having atomic join/leave operations, and more interestingly, it introduces the principles of *correction-on-use* and *correction-on-change*. Correction-on-use means that every time messages are routed, information is piggy backed to correct fingers. The more the network is used, the more accurate the routing tables become. Correction-on-change is more concerned with the detection of nodes joining or leaving the network. Every time such an event is detected, the correction of pointers is triggered immediately instead of waiting for the next round on periodic stabilization.

Atomic join/leave To solve the problem of correcting the succ and pred pointers, DKS uses an atomic join/leave algorithm [Gho06] based on distributed locks. That means that the state of a peer can be locked by another peer until it decides to release the lock. This is useful to prevent that the successor of a peer updates its predecessor pointer without the agreement of the predecessor. Providing atomic join/leave operations does not only reduce the need for periodic stabilization, but it also reduces lookup inconsistencies, which is a more important contribution. Previous attempts to provide atomic join/leave operations [LMP04, LMP06] failed to provide safety and liveness properties. The main problem was their use of three locks: succ, pred, and the joining/leaving peer. In DKS instead, only two locks are needed. To join or leave, every peer needs to get its own lock and the lock of its successor. In the case of joining this is simpler, because nobody knows about the joining peer except for itself. Then, its join operation is guaranteed as soon as it gets the lock of its successor. Leaving is relatively more difficult, because a peer cannot depart from the network if its predecessor or successor is leaving as well, and getting their locks first. We consider this an important drawback.

The algorithm guarantees safety and liveness properties. It is proven to be free of deadlocks, livelocks and starvation. However, all the proofs are given in a failure-free scenario which is unrealistic for a peer-to-peer network. If a peer crashes holding the lock of its successor, it will prevent its successor from answering lookup requests, increasing unavailability. If the peer is falsely suspected of having failed, errors can be introduced by having duplication of locks. The algorithm is also broken in presence of non-transitive connections, because peers will not be able to acquire the relevant locks to perform a join or a leave. We consider DKS's requirements too hard to meet in realistic settings. Distributed locks must not be used unless it is unavoidable, as we will see when we discuss consistent replication of storage.

Storage With respect to storage, DKS also introduces an interesting strategy to symmetrically locate replicas on the network [GOH04], instead of placing them on the successor list as Chord does. A replica is an item stored in a peer which is not the responsible of the item's key. When the responsible of the item's key is not available, a very simple data recovery mechanism can retrieve the item from any of the replicas. Symmetric replication contributes better to load-balancing and makes recovery on failure more efficient. We will discuss more about symmetric replication in Chapter 5.

Self management DKS presents very similar self-management properties to Chord but their mechanism to achieve them differ. Both rings are self-organized, self-optimized and self-healing, and peers' routing tables are self-configured. Chord achieves many of these properties through periodic stabilization and some through immediate reaction on join events and failure detection. DKS achieves self-organization through atomic join/leave algorithms. Correction-on-use provides self-optimization and self-configuration. Self-healing is achieved through correction-on-change.

2.3.3 P2PS and P2PKit

Overlay P2PS [MCV05] is also a ring-based peer-to-peer system with logarithmic routing. It is the predecessor of Beernet [MV10, MCV09, Pro09] and the relaxed ring. P2PS uses the Tango protocol [CM04] for building the finger table and routing messages across the network. Tango is very similar to Chord and DKS, but it takes into account the redundancies found in the finger tables of different nodes. One can observe that there are different paths from peer i to j with the same amount of hops, and any of those paths could be taken to resolve a lookup operation. Tango exploits these redundancies providing a more scalable and efficient solution. As drawback, it needs to take into account the information of other nodes, and therefore, it is expensive to maintain the routing tables. To compensate this cost, by exploiting redundancy, routing tables in Tango are smaller than those of DKS and Chord. The average cost for routing messages is also $O(log_k N)$ hops, but it is claimed to be 25% faster than Chord in the worst case, which is $2log_2 N$ for Chord.

Ring maintenance Taking into account the fact that two simultaneous joins with the same successor candidate created lookup inconsistencies in Chord, P2PS designed its own join algorithm. Similarly to DKS, P2PS does not use periodic stabilization to fix succ and pred pointers. It uses correction-on-change instead. Contrary to DKS, P2PS does not use distributed locks to guarantee atomic join/leave operations, which is in fact an advantage. An important contribution to fault tolerance is that graceful leaves where not considered in the design of the protocol. They are treated as failures. The reason is that if a node fails while performing a graceful-leave protocol, a failure-recovery strategy must be designed for that particular case, adding more complexity to the ring maintenance. If leave due to a failure is already handled by correction-on-change, an algorithm for graceful leaves is unnecessary.

The join algorithm of P2PS is claimed to be atomic, and in fact, it does not introduce lookup inconsistencies even if two nodes join the ring simultaneously within the responsibility range of a given peer. Even so, the algorithm relies on network transitivity to complete. Although the algorithm is atomic for two simultaneous join events, we proved in [MJV06] that the algorithm did not work in particular cases of three and more simultaneous join events, and that the inconsistency persists until the compromised peers leave the network. Unfortunately, we wrongly conclude in that technical report that a lock-based algorithm, such as the one of DKS, was needed to guarantee atomicity for join and leaves. Later, taking inspiration from P2PS's lock-free algorithm we developed the relaxed ring, which is presented in Chapter 3. As we will see, the relaxed provides atomic join without relying on transitive connectivity.

Architecture The implementation architecture of P2PS is designed in layers going bottom-up from the communication layer to more general services. Figure 2.6 is based on the architecture presented in [MCV05], and it is complemented with P2PKit's architecture, which we will soon describe. P2PS is implemented in Mozart [Moz08], which is an implementation of the Oz language [MMR95, Smo95]. The Mozart virtual machine is at the bottom of the architecture and it is accessed not only by P2PS, but also by P2PKit and the peer-to-peer application. Messages are divided into two groups: *events* and

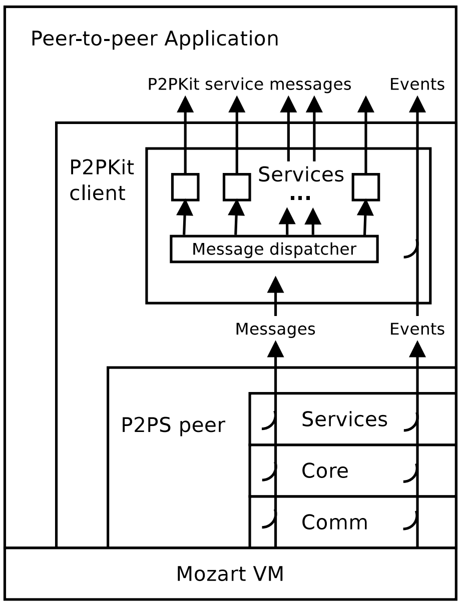


Figure 2.6: P2PS/P2PKit architecture.

messages. Events are those corresponding to the maintenance and functioning of the network, such as joins, leaves, acknowledgements, etc. Messages are those sent by the application and propagated to other peers through the network. Both sets of messages go across the three layers of P2PS, where each layer triggers new messages and new events. At the bottom of P2PS we find the `Comm` layer, which is in charge of providing a reliable communication channel between peers. The `Core` layer is in charge of the ring's maintenance, handling join and leave events, and keeping the routing table up to date. Functionalities such as general message sending, multicast, broadcast, and others, are provided in the `Services` layer.

P2PKit The API of P2PS was considered to be too basic to develop peer-to-peer applications in an easy way. The design of P2PKit [Gly05] aims to simplify the task of developers providing high-level abstractions to deploy peer-to-peer services. Although P2PKit is independent of the underlay peer-to-peer system, we present it together with P2PS because of their tight implementations [Pro08, Gly07]. Continuing with the analysis of the architecture of Figure 2.6, we observe that P2PKit creates a client separated from the peer, giving the possibility of using multiple clients for the same peer. Network events are still triggered at the application level, and new events are added by P2PKit. The crucial part of P2PKit's approach is the use of the message stream. It creates many different services as channels. These services are provided by the application working with a publish/subscribe mechanism. There is a message dispatcher in charge of filtering all messages received by the peer, putting them into the correspondent service. If the application decides not to listen to a service anymore, those messages simply will not reach the application layer as in P2PS.

Storage One of the earliest works on generic decentralized transactional protocols for replicated ring-based SONs was done on P2PS [MCGV05]. The protocol is based on two-phase locking and provides fault tolerance for the peers holding replicas. Partial fault-tolerance is provided for the transaction manager, but if the manager dies once it has taken the decision to commit the transaction, the protocol runs into inconsistencies. No implementation and no API was provided for this protocol. We will discuss again this protocol in Chapter 5.

Self management P2PS and DKS share means to achieve self-managing behaviour. Both systems rely on correction-on-change and correction-on-use to obtain self-configuration, self-optimization and part of self-healing. They basically differ in the way of handling join, leave and failure events for self-organization and self-healing. DKS attempts to provide atomic join/leave with a lock-based algorithm without handling failures very well. P2PS treats leaves and failures as the same event focusing more on fault-tolerance, using a lock-free algorithm. Both systems have problems with non-transitive connectivity.

2.3.4 Chord$^{\#}$ and Scalaris

Chord$^{\#}$ The ring-based peer-to-peer systems we have presented until now rely on the keys having a uniform distribution to balance the network load. If the keys present a different distribution, some peers will be more loaded than others, causing degradation in the performance of the system. Chord$^{\#}$ [SSR07] proposes a change on the address space and supports multiple range queries transforming the ring into a multi-dimensional torus. Regular queries retrieve items which key is in between a lower and an upper boundary. Chord$^{\#}$ has been derived from Chord by substituting Chord's hashing function by a key-order preserving function. The address space goes from characters A to Z, continuing with characters from 0 to 9. These characters are just the first of a string that determines the key. The address space is therefore infinite, circular and with total order. It has a logarithmic routing performance and it supports range queries, which is not possible with Chord. Its $O(1)$ pointer update algorithm can be applied to any peer-to-peer routing protocol with exponentially increasing pointers.

The change on the address space from integers to strings of characters can be applied to any of the previously discussed rings, therefore, it is an orthogonal issue. It can be used to provide better load-balance when it is known that the application to be developed has a non-uniform distribution of keys. Problems with non-transitive connectivity remain unsolved by Chord$^{\#}$.

Scalaris Implemented on top of Chord$^{\#}$, Scalaris provides a mature fault-tolerant and general-purpose storage service. Scalaris uses Chord$^{\#}$as a plain DHT, without using its multidimensional storage. If we just consider a plain DHT, Chord$^{\#}$'s storage is just like Chord's, except that the responsibility of every peer is infinite, although with well defined boundaries.They have implemented symmetric replication as described in [Gho06], where replicas are kept consistent with their own transactional layer [MH07]. The layer provides atomic transactions with the guarantee that at least the majority of the replicas stores the latest value of every item. The transaction would abort otherwise. We will analyse this layer in detail in Chapter 5.

Scalaris [SSR08, PRS07] uses this transactional layer over a unidimensional address space to provide a decentralized peer-to-peer version of the Wikipedia. The system has been tested on Planetlab [The03] with around 20 nodes, proving to be more scalable that Wikipedia [Wik09] itself. Wikipedia works with a server farm running MySQL database [AB95]. Their structure is centralized and not very flexible. It does not allow them to easily add new servers to the farm in case their capacity is reached. Scalaris on the contrary, thanks to its peer-to-peer architecture, can easily add new nodes to the network to increase the capacity of the service, scaling without problems.

We include Scalaris on this review because its transactional support is very close to the one implemented in Beernet. We will discuss more about their similarities and differences in Chapters 5 and 6.

2.3.5 Other SONs

In this section, for the sake of completeness, we briefly overview other SONs that have less influence in our work. We have mentioned that the contribution of the relaxed ring lies on peer-to-peer networks organized in a ring. We will see later in Section 2.5.2 that ring-based topologies tolerate better failures. The SONs included in this section use other network topologies with different guarantees. A more technical comparison is included in Section 2.5.

Kademlia [MM02, FFME04] is one of the most referenced peer-to-peer networks in literature. It assigns every peer an identifier made out of a chain of digits. Peers use a binary tree for routing where they organize other peers according to their shortest unique prefix. We believe that Kademlia works very well for networks where data does not change very much, as in file-sharing applications Emule [Emu04] and Overnet [Ove04]. But we think it would be too expensive to provide strong consistency.

Pastry [RD01a] and Tapestry [ZHS^{+}03] also use a circular address space, but we can classify them as trees because of their finger table and routing algorithm. Bamboo [RGRK04] uses the geometry of Pastry. OpenDHT [RGK^{+}05], a closely related project to Bamboo, has a nice contribution by offering a standard API for general purpose DHT. It includes security aspects to store and retrieve values from sets using *secrets*. OpenDHT already introduces the concept of key/value-sets that we will discuss in Chapter 5, but it only provides eventual consistency with no transactional support.

SONAR [SSR07] is an extension of Chord$^{\#}$ to support multi-dimensional data spaces. It uses the same geometry as CAN [RFH^{+}01], which overlay graph can be interpreted as an hypercube or a multi-dimensional torus. As we will see in Section 2.5, SONAR has a more efficient routing than CAN. However, both networks have problems handling failure recovery because their structure is too rigid. The responsibility of every peer depends on many neighbours, so it is difficult to determine the new responsible when a peer leaves the network.

Viceroy [MNR02] partitions the address space exactly as Chord. However, the routing strategy makes us classify it with a different topology, because it is based on the Butterfly graph [Mat04]. The routing table is costly to maintain, and it overloads certain peers for routing, creating congestion points.

2.4 Distributed Storage

In the first part of this chapter we have reviewed the different strategies and architectures of the most influential peer-to-peer systems for this work. In this section we discuss the different approaches used by those systems to store data. The initial and still most common goal of peer-to-peer systems is file sharing. Users share with other peers files they store locally. As soon as another peer gets the file, it becomes a *replica* and starts sharing the file with the rest of the network. The rest of the network can access any of the two replicas to get the file, and add a new replica to the network as soon as the transfer is finished. Improvements on file sharing, as in BitTorrent [PGES05], allow peers to share files divided into small chunks. Peers do not need to wait until the whole file is transferred to start sharing the chunks already downloaded. As soon as they get a chunk, they can offer their replica to the network. An even better advantage is that peers can get chunks from different peers at the same time, so the file is transferred much faster. All these protocols work very well based on the assumption that files do not change, which is a very important issue. With that assumption, any replica is a valid one, and therefore, peers need to find only one replica to get the file. This simplifies a lot of problems such as efficient routing and lookup consistency. For routing, you can follow several different paths in parallel until you find one replica. There are no consistency concerns because every replica is a valid one. This is one of the reasons why unstructured overlay networks are still popular for file sharing.

Our work considers items to be not just files, but any sort of data, and particularly application-specific data, which is constantly updated. Most applications rely on reachability and consistency of their data, which is not guaranteed on unstructured overlay networks. That is why we focus on SONs

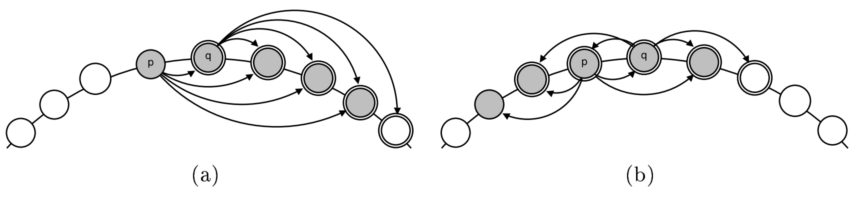


Figure 2.7: Strategies for replica placement using the neighbours of the responsible peer: (a) using the successor list, and (b) using the leaf set.

providing DHTs, with the basic `put(key, value)` and `get(key)` operations, which store and retrieve items using *key*'s responsible peer. Naturally, using only one responsible to store an item associated to a certain key is not enough to provide fault tolerance. It is necessary to provide some kind of replication. Since we are more concerned about ring-based SONs, we are going to discuss some replication techniques used in the literature in the following section.

2.4.1 Replication Strategies

Successor List There are several techniques to organize replicas on a ring. The most basic mechanism is probably the first one proposed in [SMK+01], where $log_2 N$ replicas are stored on the successor list of the responsible of each key. When a node fails, the successor takes over the responsibility, and therefore, it is a good idea that the successor stores the replicas of the items. Like that, it does not need to ask the value to other peers to continue hosting the item. This implies that each peer in the successor list must have the latest value of the item. The strategy is depicted in Figure 2.7(a). We can observe in the figure peers p and q, and their replicas stored in their respective successor lists. The replicas of peer p are shown in colour grey. The replicas of peer q are shown with a double circle. The pointers to the replica set are also added, because they are part of the resilient information that every peer needs to have, so it does not add extra connections. An important observation is that q belongs to the replica set of p, but p does not belong to q's replica set. Actually, every replica set is different.

Leaf Set A very similar strategy is the one used by networks having an overlay topology like Pastry, using the leaf-set [RD01a, RD01b] for storing the replicas. Figure 2.7(b) shows the replica set of peers p and q following this strategy. As in Figure 2.7(a), peers in grey are the replicas of p, and peers in double circle are the replicas of q. This strategy also generates a different replica set for each peer. It has the same advantages as in the successor list strategy, because it does not add extra connections, and the peer that should take over the responsibility in case of a failure already has the values of the replicas.

There are two main disadvantages with these two schemes. First of all, churn introduces many changes on the participants of the replica sets. Each join/leave/fail event introduces changes in log_2N replica sets, affecting peers that are not directly involved with the churn events. When a new peer n joins the network, it becomes part of the replica set of all peers of which n becomes a member of the successor list. This is still reasonable, but it also implies that the farthest peer on each successor list affected by n will stop being part of the replica set. In a similar way, every leave or failure will imply that a new peer needs to be added at the border of the successor list, and therefore at the border of the replica set. These changes make replica maintenance more costly. The second disadvantage is that there is a unique entry point for each replica set. To find the successor list of the responsible of a key, first, it is necessary to find who is the responsible. This means that the main peer of the replica set is a point of congestion. And if the main peer fails, first, the network needs to recover from the failure to give access to the other replicas.

Multiple Hashing CAN [RFH+01] and Tapestry [ZHS+03] use multiple hashing as replication strategy. The idea is that every item is stored with different hash functions known to all peers in the network. In Figure 2.8(a) we observe replicated items with keys i and j. The result of applying hash function h_1 to key i results in having peer p as responsible. Applying $h_2(i)$ and $h_3(i)$ gives peers a and d as responsible of the other replicas (painted in grey). Similarly, peers q, b and c (drawn with a double circle) represent the replica set of the item stored using key j. Note that replicas can be stored anywhere, and that hash function does not represent any order on the ring. One disadvantage claimed in [Gho06] is that it is necessary to know the inverse of the hash functions to recover from failures. For instance, if peer p crashes, peer a would need to know the inverse function of $h_2(i)$ to retrieve the value of i, and discover where to store the replica $h_1(i)$. This problem can be solved by also storing the original key of the item, instead of only the hash key, as it is discussed in Section 2.4.2.

A more crucial disadvantage is the lack of relationship between the replica sets per item. In the example of Figure 2.8(a), the replica set of item i is formed by p, a and d. If we take another item with key k, where the responsible peer of $h_1(k)$ is also peer p, it is very unlikely that $h_2(k)$ and $h_3(k)$ would result in hash keys within the responsibility of peers a and d. Therefore, there will be a different replica set for almost every item stored in the network, making the reorganization of replicas under every churn event very costly. When a new peer takes over the responsible of another peer, either because the other peer failed, or because the new peer joined the network as its predecessor, the new peer will have to contact the responsible peers of h_2 and h_3 of every item stored in the range involved in the churn event.

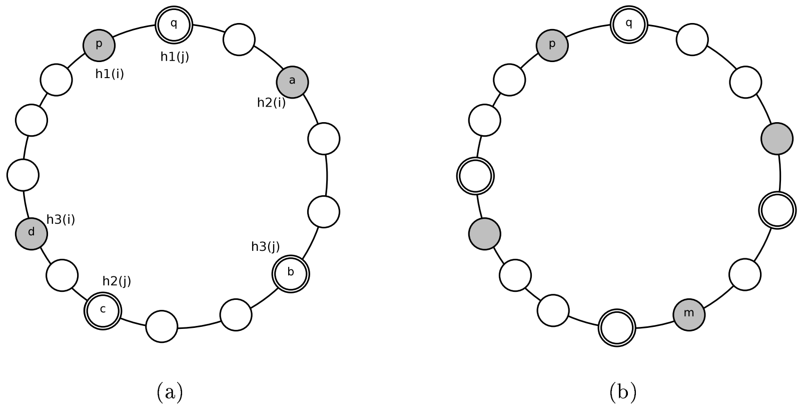


Figure 2.8: Strategies for replica placement across the network: (a) using multiple hash functions, and (b) using symmetric replication.

Symmetric Replication This is a simple and effective replication strategy presented in [Gho06] with several advantages and few disadvantages. First of all, it does not have an entry point of congestion as with the successor list and the leaf sets. Members of the replica set are not indirectly affected by churn, and as in multiple hashing, the replicas are spread across the network, with the advantage that they are symmetrically placed using a regular polygon of f sides, where f is the chosen replication factor. This strategy provides an easier way of finding the replicas, and it balances the load more uniformly.

An example of symmetric replication is depicted in Figure 2.8(b). The replicas of the items where peer p is the original responsible are painted in grey using $f = 4$ as replication factor. Replicas of peer q are drawn with a double circle as in previous examples. A small disadvantage is that it is not possible to guarantee that all replicas of all items stored on p will have exactly the same replica set. It will depend on the distribution of the address space amount the nodes. Even though this is guaranteed in strategies using the successor list or the leaf sets, there is another advantage of symmetric replication that overcomes this drawback. As we saw in the analysis of the successor list, every peer has a different replication set. In symmetric replication, it is possible that f peers share the same replication set in the ideal case, and in real cases, they will share most of the replicas with the same peers. In Figure 2.8(b) we have added peer m as if it were an ideal case. In this example, the replica set of peer m is exactly the same as the one of peer p, where every peer stores the replica of the other members of the set. This property cannot be guaranteed for all keys, but it minimizes enormously the amount of nodes that a peer needs to contact to recover from a failure.

A disadvantage shared by multiple hashing and symmetric replication is that both rely on a uniform distribution of peers in the address space. However, this assumption is very reasonable since many SONs also rely on this property to achieve the promised logarithmic routing. In case of very skewed distributions, one could observe that one peer is the responsible of two replica keys, decreasing the size of the replica set, and therefore, decrease the fault tolerance factor. This is less probable in symmetric replication, and the larger the network, the less probable this event is to happen, so it scales up without problems.

2.4.2 How to Store an Item

One of the most basic operations offered by a DHT is `put(key, value)`, where a hash function $h(key)$ is used to determine the *hash key* to find the responsible of the item to be stored in the network. But how is the item stored in the peer? Let us consider the example of the operation `put(foo, bar)`. Then, let us say that $h(foo) = 42$, then, most networks assume that the item to be stored is $i = (42, bar)$. That is why in [Gho06] it is claimed that you need the inverse hash function to work with multiple hashing as replication strategy. Another problem is that the diversity of keys that can be used is limited by the chosen size of the address space. If $N + 1$ keys are used in an address space of size N, there will be at least two keys having the same hash key, meaning that one of the two values will be lost. Choosing a very large value of N will unnecessarily result in a large routing table based on the value of N, and it still represents a limit on the maximum amount of keys to be used.

A better way of storing an item, however more costly, it is to store the key and the hash together with the value. In that way, our `put(foo, value)` operation would result in storing the item $i = (42, foo, bar)$. If another operation has the same *hash key*, say `put(alice, bob)`, with $h(alice) = 42$, it would simply add item $j = (42, alice, bob)$. The chosen value of N would not limit the amount of keys to be used, but only the amount of peers that can join the network. This simple analysis is actually often omitted by several network descriptions, but as we can see, it can have some implications on the need for inverse hash functions.

A third option would be to simply store the key and the value, and use the hash key only to find the responsible for the storage. Following that approach, the operation `put(foo, value)` would result in storing the item $i = (foo, bar)$ on the peer responsible for key $h(foo) = 42$. The problem with this strategy is that ranges of responsibility constantly change due to churn. When a peer gets a new predecessor, its range its split in two, and several items will need to be transferred to the new predecessor. To know which items belongs to the predecessor's responsibility, the hash key of all items would need to be recompute, making churn less efficient. Another consequence of not storing the hash key on the item, is that peers would need to recompute the hash key every time they need to know if an item belongs to its responsibility range, or if it is a replica.

In our view, we consider that the strategy of storing the item with its key and hash key is the most convenient, because it does not require inverse hash functions and it prevents conflicts when two keys have the same resulting hash key. Therefore, this is the way Beernet store items.

2.4.3 Transactions

A common architecture decision for building DHTs is to organize the functionalities in bottom-up layers. The overlay graph maintenance together with lookup resolution is placed at the bottom layer. The DHT functionalities `put` and `get` are normally built on top of the bottom layer. Replica management, with the chosen replication strategy as we discussed in Section 2.4.1, is a layer built on top of the basic DHT. We will continue discussing this architecture in the following chapters but now we will say a word about keeping the replicas consistent. There are basically two choices: adding the consistency maintenance at the replica layer, or build a transactional layer on top of it that can guarantee not only that the replica set of an item is kept consistent, but also that the replica set of several items is kept consistent in an atomic operation.

We will discuss transactions in detail in Chapter 5, but first we will review what has been done in the field. The transactional layer has the goal of providing the ACID properties to data storage on the DHT. This goal also holds even if the transaction is applied to only one item. ACID properties concern: Atomicity, Consistency, Isolation and Durability. Orthogonally to the decision of where to store the replicas, the main issue to solve is how to manage the update of the replicas to provide a consistent access to the state. The classical approach is Two-phase-commit, which is not suitable for peer-to-peer because it presents a single point of failure. This problem can be overcome using replicated transaction managers, and the Paxos consensus algorithm where the majority of the replicas decides on the update of the value of an item. Another alternative is three-phase commit, which even though it uses less messages per round, it introduces an extra round to the protocol which is undesirable on peer-to-peer networks. Later on we will discuss the validation of such algorithms, and how Paxos consensus can be adapted to provide a more eager way of performing a distributed transaction. This is optimistic vs pessimistic approach.

Ivy [MMGC02] is one of the earliest works having a transaction-like system for distributed storage. It is built on top of Chord [SMK$^+$01], and it is based on versioning logs per peer. Updates on the replicas do not guarantee consistency, but the log information is meant to be useful for conflict resolution. A more complete and fully transactional protocol [MCGV05] was designed for P2PS [MCV05], but it was never implemented. This protocol was based on two-phase commit having the problem of relying on the survival of the temporary transaction manager to complete the transaction. The goal of their protocol was to show that it was feasible to build decentralized transactions. The Paxos consensus algorithm was presented in [MH07] and implemented in Scalaris [SSR08] and Beernet [MHV08].

Table 2.1: Comparing unstructured and structured overlay networks.

	Unstructured	Structured
Topology	Random	Ring, Hypercube, Tree, etc.
Routing	Flooding	Directed using meaningful routing tables
Guarantees	Lookup converges most of the time	Routing cost is bound, mostly logarithmic. All peers are reachable
Provides	File sharing	DHT
Transitive connectivity	Not needed	It relies upon it

2.5 Summary of Overlay Networks

In previous sections of this chapter, we have described the main features, advantages and disadvantages of several overlay networks. In this section we summarize them trying to apply the same criteria to all of them. For the sake of readability, sometimes we will present the information in tables, and sometimes as lists.

2.5.1 Unstructured and Structured Overlays

As we already presented it, most peer-to-peer networks can be classified into three different generations. The first one still relied on a hybrid architecture where servers were needed to bootstrap any peer-to-peer service, and peers were not able to route messages. We are more interested in the second and third generations, also called unstructured and structured overlay networks, both of them being completly decentralized and able to self-organize. Table 2.1 shows a summary of the main features of these two generations. Both of them use the most suitable routing strategy according to the overlay topology used to organized the peers. SONs provide stronger guarantees, and by providing a DHT, file sharing is also possible, and not only for popular items, because reachability is also guaranteed. An interesting issue related to the topology used is the fact that unstructured networks do not need transitive connectivity, whereas SONs rely upon this property which cannot be guaranteed in all cases on the Internet, basically due to latency and NAT devices, as we discussed in the introduction of Section 2.3. The fact that the overlay topology is more relaxed in unstructured networks provides this advantage, with the cost that lack of structure prevents the provision of strong guarantees. This conclusion will become very important when we motivate the relaxed ring in Chapter 3.

Table 2.2: Properties of structured overlay networks.

Network	Overlay	Routing
Chord, Chord$^{\#}$, OpenDHT	Ring	$O(log_2 N)$
DKS, P2PS	Ring	$O(log_k N)$
Pastry, Tapestry	Tree	$O(log_{2^b} N)$
Kademlia	Tree	$O(log_2 N)$
CAN	Multidim. Torus	$O(\sqrt[d]{N})$
SONAR	Multidim. Torus	$O(log_d N)$
Viceroy	Butterfly graph	$O(log_2 N)$

2.5.2 Structured Overlay Graphs Comparison

We now have a look at the features of the structured overlay networks described in Section 2.3. Table 2.2 summarizes their overlay graph and the complexity of their routing cost. The routing cost considers only the amount of hops a lookup request needs to reach the responsible of a key. It does not considering the total amount of messages sent. For instance, in Kademlia, the lookup request follows several paths in parallel until it reaches one of the replicas. But, the table only considers the amount of hops of the successful path. We have classified Pastry, Tapestry and Kademlia as trees, because the graphs defined by the fingers and leaf sets form a tree. However, they also use a circular address space organizing peers in a ring. This means that failure handling is similar to the networks we have classified as rings, with the addition of the extra updates that are needed on the leaf sets.

We have classified Chord$^{\#}$ as a ring because it is its most common use, although it can support multi-dimensional data spaces. We can observe in Table 2.2 that almost all networks guarantee logarithmic routing, except for CAN, even though its complexity is also very good. The base of the logarithm is the main difference between networks, and it is directly influenced by the election of the routing table. Our conclusion is that in general, structured overlay networks are very competitive in terms of routing.

Concluding remark With respect to fault tolerance, failures in a ring mainly affect the successor and predecessor of the failed node. Fingers are also affected but this only comprises efficiency, not correctness. Failures in other overlays, such as trees, hypercubes or butterfly graphs imply changes in a lot more peers that need to reorganize the overlay graph. Adding nodes to these structures can be done very efficiently, but removing peers is very costly. Therefore, we arrive to the same conclusion given in [GGG^{+}03], that ring-based networks are competitive in routing cost, but they are more tolerant to failures.

Table 2.3: Failure handling on ring-based overlay networks

Network	Mechanism	Leaves and Failures
Chord, Chord$^{\#}$	Periodic stabilization	Leave = Failure
DKS	Correction-on-change	Gentle leave is fundamental
P2PS	Correction-on-change	Leave = Failure

2.5.3 Ring Based Overlays

We focus now on the analysis of the different peer-to-peer networks built using a ring as overlay graph. We know that Chord and Chord$^{\#}$ rely on periodic stabilization to fix successor, predecessor and finger pointers. Such strategy has the advantage of treating leaves as failures. Therefore, there is no need to define a protocol for gentle leaves, because pointers will be fixed in the next round of stabilization. However, this implies that stabilization needs to be run often enough, increasing bandwidth consumption. It has been shown in [Gho06] that lookup inconsistencies can appear in Chord just because of churn, even if failures do not occur. This is a serious problem for correctness. To avoid this problem, DKS introduces the concept of correction-on-change, meaning that pointers are fixed as soon as a failure, leave or join is detected. Peers do not wait for a periodic check.

DKS defines a protocol for atomic join/leave to avoid lookup inconsistencies. The protocol requires that a peer respects the locking protocol before it leaves the network. Unfortunately, this strategy is not very fault-tolerant, and it relies on peers not leaving the network before they acquire the needed locks. P2PS also addresses ring maintenance with correction-on-use, but it does not require gentle leave to fix the departure of peers. This approach allows P2PS to solve the problem of leaving peers by handling peer failures. A peer that leaves the network could send messages to provide a more efficient fix of pointers, but it is not needed for correctness. The problem with P2PS is definitely not the approach, but the protocol does not work in some cases, as it is shown in [MJV06]. A more fundamental issue, also shared by the other networks, is that P2PS relies on transitive connectivity, and it expects to create a perfect ring with respect to successor and predecessor pointers. Table 2.3 summarises these differences, but it only talks about fixing successor and predecessor pointers. Fingers are also solved with periodic stabilization in Chord and Chord$^{\#}$, but they are addressed with correction-on-use in DKS and P2PS.

Concluding remarks The loosely join algorithm of Chord creates lookup inconsistencies even in failure-free environments. Periodic stabilization is an expensive approach to fix this problem. A cost-efficient approach is the atomic join introduced by DKS using distributed locks. But this approach relies on requirements that are too hard to meet in realistic scenarios. P2PS offers a good compromise between cost-efficiency and a relaxed join algorithm, however, it shares the same problems as the other networks: it requires a perfect ring and relies on transitive connectivity. Both properties being difficult to guarantee.

2.6 Analysis of Self-Management Properties

In this dissertation we claim that self-management is essential for building dynamic distributed systems. We mentioned six different axes to analyze self-management, and we identify them in the overlay networks we have reviewed in this chapter. Since several of these axes are shared by many of the networks, with some small differences, we will not build a comparative table, but we will summarize what we have observed for each axis.

Self organization This is the property that it is most intrinsically present in the overlay networks we have reviewed, even in networks such as Gnutella and Freenet, where self-organization is very basic considering the lack of structure. Self organization comes naturally as there is no central point of control, and peers needs to agree their position only by talking to the direct neighbours. The only pre-existing infrastructure needed to boot the system is the Internet, or any underlying network that provides means to establish point-to-point communication channels. None of these networks have human intervention that assign peers their position in the network. Everything is self-organized. Table 2.3 gives us hints about some of the mechanisms used to achieved self-configurations. Networks such as Chord and Chord$^{\#}$ use periodic stabilization to keep the ring organized. DKS and P2PS use correction-on-change for the same purpose.

Self configuration First of all we need to clarify what is that we want to configure. This property is usually used to mean self-configuration of components. But that would imply already a design decision of implementing the system using components. Although that would be a very good decision, the networks we have reviewed could be implemented in many different ways. We will discuss self-configuration of components in Chapter 7. Here, we are interested in the configuration of the overlay graph. Someone could argue that this is already covered by the property of self-organization, but, if we compare Chord with Pastry, both are organized as rings, but their overlay graphs are configured differently, making Pastry work as a tree. What needs to be configured is the routing table of each peer, which could actually be seen as a component. Self-configuration is achieved in Chord, Chord$^{\#}$, and the majority of the networks, with periodic stabilization. Peers constantly check the status of the neighbours of their fingers, and update the routing table according to this information. Similarly to self-organization, DKS and P2PS use correction-on-change to re-configure their routing tables, but also correction-on-use, piggy-backing configuration messages to the regular messages that are routed through the network, without increasing unnecessarily bandwidth consumption.

Self healing In the context of peer-to-peer networks, this property deals with handling disconnection between peers. Disconnection can be produced not only by the failure of peers, but also due to problems in the communication channel between them. Because of this, it is very important to know if the protocols for failure handling rely on perfect failure detection or not. As we mentioned in the Introduction chapter, Internet style failure detectors are typically strongly complete and eventually accurate, therefore, eventually perfect. As a consequence, an expensive failure recovery mechanism can become very impractical if the network is to be built on top of the Internet. For instance, the failure recovery might be cancelled if a node is falsely suspected of having crash, and then it is detected of being alive. As we conclude earlier in this section, ring-based overlay networks can handle failures more efficiently. Not surprisingly, Chord-like rings achieve self-healing with periodic stabilization. If a node disappears from the network, ring pointers will be fixed in the next stabilization round. If it was a false suspicion, the incorrectness will be fixed in the stabilization round after the accuracy is achieved. P2PS and DKS rely on correction-on-change to provide self-healing. This approach requires that the failure detector triggers the corresponding *crash* and *alive* events to react to failures and false suspicions. DKS has the drawback of relying on gentle leaves of peers to respect the locking protocol providing atomic join/leave. What is missing from all the networks we have reviewed, is a correct handling of non-transitive connections, which can be seen as permanent false suspicions. For instance, if node a receives from node b a reference to node c, but it cannot establish a communication with it, a will suspect that c is dead, however, b can always communicate with c. We will see later on this dissertation how Beernet can cope with this problem.

Self-healing is not only about fixing pointers on the overlay graph, it also covers repairing stored items. We discussed that plain DHT was not enough to guarantee fault tolerance, and therefore, some replication mechanism was needed. We described several strategies to place replicas, such as leaf sets or symmetric replication. The mechanism that triggers self-healing in any of these strategies is orthogonal to the mechanism for fixing the overlay graph pointers. For instance, Chord$^{\#}$ and DKS can implement symmetric replication having the same self-healing mechanism to restore replicas, but they have different mechanism to fix the overlay graphs, namely periodic stabilization and correction-on-change.

Self tuning This topic is not really discussed by the designers of the networks we have studied, but we can identify variables whose values can be tuned to provide a better behaviour. Periodic stabilization is a clear example where self-tuning makes sense, because the frequency of each stabilization round can be too short, consuming too much bandwidth. Or it can be too long, having always dead pointers that have not been updated, breaking the guarantees the network provides, as it is shown in [KA08]. Other variables than can be tuned are those involved in failure detection: how often a peer needs to be pinged? Or how long does the timeout has to be? These parameters will affect how fast correction-on-change will react. Evidently, since every network needs failure detection, tuning these parameters will not help only P2PS and DKS, but any network we have described here.

Self optimization One of the most important values to optimize in peer-to-peer networks is routing cost. It is very important that fingers are constantly updated to work with routing tables that are optimized. Therefore, there is some overlap with self-configuration, because both properties actuate over the routing tables. As we mentioned before, DKS and P2PS clearly provide self-optimization by means of correction-on-use. The more the network is used to route messages between peers, the more performant it becomes.

Self protection The only network we have presented in this section that explicitly treated security issues is OpenDHT with its API prepared for storing data with encryption. However, this API does not provide self-protection, because once the keys are compromised, there is no mechanism to detect and repair the encrypted storage. One of the few works on self-protection we are aware of in the scope of peer-to-peer networks, is the study of Small world networks [HWY08] to provide self-protection against the Sybil attack [Dou02]. The authors claim that an overlay graph built as small world network is the only solution to the Sybil attack, and therefore, rings, trees or hypercubes, are subject to such attack. In our view, the work is actually based on the good properties of social networks that can identify when an attacker is trying to pretend being different persons to gain reputation. Social networks typically build small world graphs, but the self-protection does not come from the graph, but from the fact that real people are behind the nodes participating in the network.

2.6.1 Scalability

With respect to scalability, there are no major differences between the different structured overlay networks. Almost all of them scale well to very large networks. Systems based on Butterfly graphs could have some problems considering the bottle neck presented on the highest layer of routing, but the larger the network becomes, more peers will be placed at the highest layer, balancing the load of routing messages. Still, the bottle neck we previously identified remains an issue. What is interesting to observe is that none of these networks seems to be suitable to scale down. In small peer-to-peer networks one could benefit from full mesh networks, instead of using sophisticated routing tables to provide logarithmic routing. This dissertation also makes a contribution in this topic with a self-optimized routing table that is able to scale up and down, a property that can also result interesting in Cloud Computing, as we will discus in Section 2.8.

2.6.2 Replicated Storage

While discussing self-healing, we mentioned that several strategies could be adopted to provide storage replication. A summary of the four replication strategies described in Section 2.4.1 is presented here, based on where the replicas are stored, which network implements the strategy, what is the semantic of the replica set, advantages and disadvantages.

- **Successor List**.
  - **Implemented** in Chord.
  - **Replica set** per peer. Placed in the successor list.
  - **Advantages** In case of failure, the successor has already the replicated values.
  - **Disadvantages** To find the replicas, the responsible of the hash key needs to be found first. There is a central point of congestion. Churn affects border members of the set.
- **Leaf Set**.
  - **Implemented** in Pastry.
  - **Replica set** per peer. Placed in $f/2$ successors + $f/2$ predecessors.
  - **Advantages** Same as successor list, plus, since some replicas can be found earlier because they are placed on the predecessors.
  - **Disadvantages** Same as successor list.
- **Multiple Hashing**.
  - **Implemented** in CAN and Tapestry.
  - **Replica set** per key. Placed on the responsible of every hash function.
  - **Advantages** Spread across the network. No point of congestion.
  - **Disadvantages** Too many replica sets. Depending on the storage, the inverse hash function might be needed.
- **Symmetric Replication**.
  - **Implemented** in Chord$^{\#}$ and Beernet.
  - **Replica set** per key but approximately organized in sets of f peers. Placed symmetrically across the network.

– **Advantages** Load balancing of replicas symmetrically spread across the network. Any replica can be accessed without having to contact the main responsible. Most of the time one replica set groups several items and f peers, making groups more closely connected. No point of congestion. No need for inverse hash function.
– **Disadvantages** It relies on uniform distribution of peers on the address space.

2.7 A Note on Grid Computing

Peer-to-peer networks and Grid computing have the common goal of sharing resources and make them available to their users. Both systems need to locate the resources coming from different heterogeneous processes. Despite these similarities, the approaches differ in several aspects due to their different characteristics. Peer-to-peer is conceived to allow collaboration among untrusted and anonymous users, whereas Grid users are typically trusted and identified members from research institutions or known organizations within a federation. Peer-to-peer is fundamentally decentralized and can scale to very large networks. Grid networks on the contrary, are comparatively smaller and do not scale because Grids are mainly built gathering sets of clusters from well connected organizations. This means that there is almost no churn and failure detection is much more accurate. Such scenario has allowed Grids to be designed as centralized and hierarchical. However, Grids are becoming larger and larger, and the centralized approach will have to leave its place to decentralized self-management. As mentioned in [FI03, TT03], Grid computing and peer-to-peer will eventually converge. There is already research that targets Grid computing from a peer-to-peer approach [IF04, TTZH06a, TTZH06b, TTF$^+$06, MGV07, TTH$^+$07], providing fault-tolerance, resource discovery, resource management and distributed storage. These works indicate us that the results from this dissertation can be applied to Grid computing if the design is merged with peer-to-peer systems. Even so, our research is orthogonal to Grid.

2.8 A Note on Cloud Computing

Cloud computing is one of the latest emerging research topics in distributed computing, and therefore, it is necessary to contextualize it in this dissertation. Cloud computing has many definitions with different views within industry and academia, but everybody agrees on that cloud computing is the way of making possible the dream of unlimited computing power with high availability. Cloud computing has been active in the IT industry for a couple of years already, calling immediately the attention of the research community thanks to its possibilities and challenges. Projects such as Reservoir [Res09], XtreemOS [Par09] and OpenNebula [Dis09] are just examples of the interest of the research com-

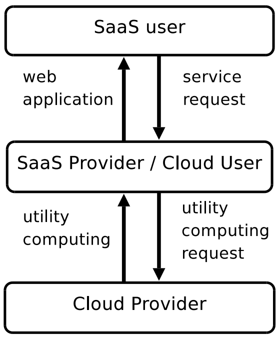


Figure 2.9: General Cloud Computing architecture.

munity. However, defining Cloud computing it is not that simple. One of the interpretations sees Cloud computing only focused on *high availability* and on the idea that computations are mainly done elsewhere, and not on user's machine. Another view considers any application provided as a web service to be *living in the cloud*, where the cloud is simply the Internet. We share Berkeley's view of Cloud Computing [AFG$^+$09] and the conclusions of 2008 LADIS workshop [BCvR09]. We see Cloud Computing as the combination of hardware and software that can provide the illusion of infinite computing power with high availability.

Large companies provide this illusion of infinite computing power by having real large data centers with software capable of providing access on demand to every machine on the data center. Industrial examples supporting Cloud Computing are Google AppEngine [Goo09], Amazon Web Services [Ama09] and Microsoft Azure [Mic09]. These three companies follow the architecture described in [AFG$^+$09] where the base of the whole system is such a large company being the cloud provider. Cloud users are actually smaller companies or institutions that use the cloud to become Software as a Service (SaaS) providers. The end user is actually a SaaS user, which is indifferent to the fact that a cloud is providing the computational power of the SaaS.

Figure 2.9 depicts the general architecture described in [AFG$^+$09]. The *cloud provider* is at the base of the architecture offering *utility computing* to the *cloud user*. Utility computing can be understood as a certain amount of resources during a certain amount of time, for instance, a web server running for one hour, or several Tera bytes of storage for a certain amount of days. The cloud user, which is actually a *SaaS provider*, has a predefined utility computing request, which can vary enormously depending on its users demands. At the top of the architecture we find the *SaaS user* which requests services from the SaaS provider. The service that the SaaS provider offers to its users is usually presented as a web *application*.

Given such architecture, there are two parts that might be relevant to this dissertation: the cloud provider and the SaaS provider. First of all, it is necessary to build a network capable of managing the resources at the bottom of the architecture. XtreemOS [Par09] focuses on that part, and sees Cloud computing as an extension to Grid computing. OpenNebula [Dis09] not only allows the management of the local cloud infrastructure, but it also abstracts the cloud provider introducing an interface layer to the SaaS provider. Like this, the SaaS provider could choose to manage its own infrastructure, or hire a large cloud provider such as Amazon. The interesting thing here is that once the cloud provider is abstracted, it raises the possibility of using multiple cloud providers behind the interface layer. The interface layer would then work as a resource broker that decides to which provider the request will be sent. Like this, a company can rely only on its own resources until a peak in users' demand appears, and then, it hires some extra resources from Amazon. Schemes like this are starting to be developed by projects such as Reservoir [Res09] and Nimbus [The09b]. In the later, they also talk about *sky computing environment* where several clouds are constantly providing the needed resources. A very similar idea is presented as work in progress in XtreemOS under the names of *community cloud* and *cloud federation*, making the difference on who is providing the resources. Having several cloud providers behind the interface layer will make the use of a centralized resource broker unusable considering fault tolerance and scalability, giving room for research on decentralized systems.

From the point of view of the SaaS provider, it has to be able to request and release resources according to the demand of its SaaS users. In other words, its own network must quickly scale up and down to maximize quality of service, and to minimize costs. This looks as a network with controlled churn which might be interesting to investigate as a peer-to-peer network.

2.9 Conclusion

This chapter summarizes large part of the work that has been done in decentralized networks. We went through the three generations of peer-to-peer networks, making a deep analysis of structured overlay networks, known as the third generation. We still identify interesting properties from the second generation, represented by unstructured overlay networks. We have reviewed them in section 2.2.3, analysing how they where able to build a completely decentralized system where peers did not need any central point of management to organize themselves. Although unstructured overlay networks are successful for building file-sharing services, their applicability is quite limited to that. This is, among other reasons, due to their lack of guarantees of reachability and lookup consistency. SON is the attempt to provide these properties in peer-to-peer networks, and the way to achieve them is by giving structure and increasing self-management.

From all SON topologies, we have observed that all of them are very competitive in terms of routing complexity, and that ring-based networks performed better in terms of failure recovery. We have also analyzed the networks using six axes of self-management: self-organization, self-configuration, self-healing, self-tuning, self-optimization and self-protection. We have identified that within ring networks there are two main tendencies to achieve several of the self-management properties. Chord-like networks based their network management on periodic stabilization, whereas DKS and P2PS use correction-on-change and correction-on-use, have a more efficient bandwidth consumption.

We have also discussed the problem that nearly all protocols we have reviewed rely on transitive connectivity. It has been observed even by the authors of Chord, Kademlia and DHT, that non-transitive connectivity generates several problems when these networks have been deployed as real systems running on top of the Internet. The lesson learned was that non-transitive connectivity should be taken into account from the design of the network. We also add imperfect failure detection as one of the issues that needs to be considered in the design. We also noted that transitive connectivity is not really an issue in Gnutella or Freenet, partly because of their relaxed overlay network.

The most characteristic feature of SONs is that they provide DHTs. Being aware that plain DHT support is not enough to provide storage fault tolerance, we showed several replication strategies in Section 2.4. Two of them try to take advantage of the overlay graph placing replicas either on the successor list or on the leaf set. The disadvantages are that in both cases there is a single entry point that creates a bottle neck, and that churn generates a lot of traffic associated to replica maintenance. To balance the load of replica storage, we have analyzed multiple hashing and symmetric replication. This last one is the simplest one to implement and minimizes the differences between related replica sets. Therefore, symmetric replication appears as the most convenient replication strategy.

Finally, we have briefly reviewed how peer-to-peer networks are related to Grid computing and Cloud Computing. Basically, peer-to-peer has mainly been used to implement resource-discovery services on the Grid but they are rarely used as overlay network to organize them. There is no direct connection to Cloud Computing as it is now, having its business model based on a single large cloud provider. We believe that the new academic tendency to conceive Cloud Computing with several small cloud providers will give a chance to decentralized systems and peer-to-peer networks to contribute to the self-management of the system.

Chapter 3

The Relaxed Ring

One ring to rule them all
One ring to find them
One ring to bring them all
and in the network bind them

freely adapted from *"The Lord of the Rings"* -
J.R.R. Tolkien

Chord [SMK$^+$01] is the canonical structured overlay network (SON) using ring topology. Its algorithms for ring maintenance handling joins and leaves have been already studied [Gho06] showing problems of temporary inconsistent lookups, where more that one node appears to be the responsible for the same key. Peers need to trigger periodic stabilization to fix inconsistencies. Existing analyses conclude that the problem comes from the fact that joins and leaves are not atomic operations, and they always need the synchronization of three peers, which is hard to guarantee with asynchronous communication, which is inherent to distributed programming.

Existing solutions [LMP04, LMP06] introduce locks in the algorithms to provide atomicity of the join and leave operations, removing the need for a periodic stabilization. Unfortunately, locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free environments, which is not realistic. Another problem with these approaches is that they are not starvation-free, and therefore, it is not possible to guarantee liveness. A better solution using locks is provided by Ghodsi [Gho06], using DKS [AEABH03] for its results. This approach is better because it gives a simpler design for a locking mechanism and proves that no deadlock occurs. It also guarantees liveness by proving that the algorithm is starvation-free. Unfortunately, the proofs are given in fault-free environments and assume full connectivity between nodes.

The DKS algorithm for ring maintenance goes already in the right direction because it request the locks of only two peers instead of three (as in [LMP04, LMP06]). More details about how it works are given in Section 2.3.2. One of the problem with this algorithm is that it relies on peers gracefully leaving the ring, which is neither efficient nor fault-tolerant. The algorithm becomes very slow if a peer holding a relevant lock crashes.

As we have already discussed in Chapter 2, the fundamental problem that affects all these algorithms is that they all rely on transitive connectivity, a property that is not guaranteed by Internet style networks. The King data set [GSG02] is a matrix of latency measurements between DNS servers. It is often used to test peer-to-peer protocols with realistic values of network delays. In this matrix, we observe that 0.8% of the links present a value of -1, meaning that the communication was not established between the DNS servers, introducing non-transitive connectivity. The authors of Chord, Kademlia and OpenDHT described in [FLRS05] that their measurements on PlanetLab [The03] encountered 2.3% of all pairs having problems to communicate. The conclusion is that non-transitive connectivity must be taken into account from the design of a decentralized system, and this is what the relaxed ring pretends to do. They create a ring-based network with good properties on lookup consistency, despite the existence of non-transitive connectivity. In Chapter 6 we will discuss the behaviour of the relaxed ring running with different amount of peers running behind NAT devices.

3.1 Basic Concepts

As in any overlay network built using ring topology, in our system, every peer is identified with a key. Every peer has a successor, a predecessor, and a routing table formed by so called fingers. Fingers are used to forward messages to other parts of the ring providing efficient routing. Ring's key-distribution is formed by integers from 0 to $N-1$ growing clockwise, where N is the size of the address space. When a peer receives a message, the message is triggered as an event in the ring maintenance component. The range between keys, such as $]p,q]$, follows the key distribution clockwise. There is one case where $p>q$, being p the predecessor of q. In such case, the range goes from p to q passing through 0.

As we previously mentioned, one of the problems we have observed in existing ring maintenance algorithms is the need for an agreement between three peers to perform a join/leave action. We provide an algorithm where every step only needs the agreement of two peers, which is guaranteed with a point-to-point communication. In the specific case of a join, instead of having one step involving three peers, we have three steps involving two peers. Lookup consistency is guaranteed after every step, therefore, the network can still answer lookup requests while simultaneous nodes are joining the network. Another relevant difference is that we do not rely on graceful leaving of peers. We treat

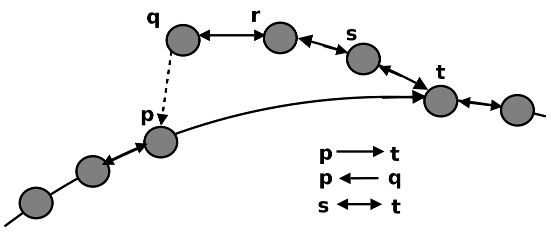


Figure 3.1: A branch on the relaxed ring created because peer q cannot establish communication with p. Peers p and s consider t as successor, but t only considers s as predecessor.

leaves and failures as the same event. This is because failure handling already includes graceful leaves as a particular case.

Normally the overlay is a ring with predecessor and successor knowing each other. If a new node joins in between these two peers, it introduces two changes. The first one is to contact the successor. This step already allows the new peer to be part of the network through its successor. The second step, contacting the predecessor, will close the ring again. Following this reasoning, our first invariant is that:

Every peer is in the same ring as its successor.

Therefore, it is enough for a peer to have connection with its successor to be considered inside the network. Secondly, the responsibility of a peer starts with the key of its predecessor, excluding predecessor's key, and it finishes with its own key. Therefore, our second invariant is that:

A peer does not need to have connection with its predecessor, but it must know its predecessor's key.

These are two crucial properties that allow us to introduce the relaxation of the ring. When a peer cannot connect to its predecessor, it forms a branch from the *"perfect ring"*. Figure 3.1 shows a fraction of a relaxed ring where peer t is the root of a branch, and where the connection between peers q and p is broken.

Having the relaxed ring architecture, we use a principle that modifies Chord's routing mechanism. The principle is that *a peer p always forwards the lookup request to the responsible candidate*. This mechanism will prevent that p misses any peer in between its successor and itself. It also works if p is the real predecessor of the responsible. This principle is strongly related to the concept of *local responsibility* defined in [SMS+08]. It is also presented as a patch [FLRS05] to deal with non-transitive links in PlanetLab.

Using the example in Figure 3.1, p may think that t is the responsible for keys in the interval $]p, t]$, but in fact, there are three other nodes involved in this range. In Chord, p would just reply t as the result of a lookup for key q. In the relaxed ring, peer p forwards the message to responsible candidate t. When the message arrives to node t, it is sent backwards to the branch, until it reaches the real responsible. Forwarding the request to the responsible is a conclusion we have already presented in [MV07], and it has been recently confirmed by Shafaat [SMS+08].

Introducing branches into the lookup mechanism modifies the guarantees about proximity offered by Chord. To reach the root of a branch it takes $O(log_k(n))$ hops as in DKS (in Chord, $k = 2$), because the root of the branch belongs to the *core-ring*. Then, the lookup will be delegated a maximum of b hops, where b corresponds to the distance of the farthest peer in the branch. Then, lookup on the relaxed ring topology corresponds to $O(log_k(n) + b)$. We will see in Chapter 6 that the average value b is smaller than 1 for large networks.

Before continuing with the description of the algorithms that maintain the relaxed ring topology, let us define what do we mean by lookup consistency.

> **Def.** *Lookup consistency means that at any time there is only one responsible for a particular key k, or the responsible is temporary unavailable.*

3.2 Bootstrapping

Every network bootstraps from a single node having itself as successor and predecessor. We can say that every newly created node belongs to its own ring. Algorithm 1 describes the set of variables a peer in the relaxed ring should know. The node is created with a key to become its own identifier, which is generated from a random key-generator from 0 to N. For simplicity, we will use this key not only as identifier but also as connection reference. For real implementation, every peer needs a key identifier and a network reference.

As we already mentioned, the node becomes its own successor (*succ*) and predecessor (*pred*), meaning that it is alone in its own ring. For resilient purposes, the node uses a successor list (*succlist*) which looks ahead on the ring, having a maximum of $log_k(N)$ peers. We will explain later the use of this list. The predecessor list (*predlist*) is not the equivalent backward to the successor list. It contains all peers that consider the current node as its successor. Initially, these two lists contain the node itself as part of the bootstrapping procedure. The algorithms presented in this chapter consider these lists as clockwise-sorted sets, having the identifier of the node as reference point. We will discuss these two sets later. The finger table used for routing works exactly as DKS. Chord is a special case of DKS where $k = 2$. We will use *fingers* during the description of our routing strategy. However, we will not include it in the description of the ring maintenance, because it is an orthogonal component taken from the literature, and it has been discussed already in Chapter 2.

Algorithm 1 Bootstrapping a peer in the relaxed ring

```
procedure init(key) is
    self := key
    succ := self
    pred := self
    succlist := {self}
    predlist := {self}
end procedure
```

Note that Algorithm 1 is a simple procedure. During the rest of the chapter, we will also use the *event-driven notation* to describe algorithms. In the event-driven model, every event is handled in a mutually exclusive way, avoiding share-state concurrency within every peer. An event represents an incoming message, which is remotely triggered with the operator **send**. The send operation is asynchronous, non-blocking and it does not trigger an exception if the remote process is not alive. We will discuss more about the event-driven model in Chapter 7. The following example shows a peer ('self') answering *pong* to an incoming *ping* message (from peer 'src').

```
upon event ⟨ ping | src ⟩ do
    send ⟨ pong | self ⟩ to src
end
```

3.3 Join Algorithm

To be able to join another ring, a peer needs an access point, that can be any peer on the ring. The new peer triggers a lookup request for its own key to find its best successor candidate. This is a quite usual procedure for many SONs. When the lookup request reaches the responsible of the key, the responsible contacts the new peer with the event *replyLookup*. Finding the responsible of a key can have different uses depending on the application, but in this case we will just use it to trigger the join process. The routing protocol to reach the responsible will be explained in the following section, because we first need to understand how branches can be created during the joining process.

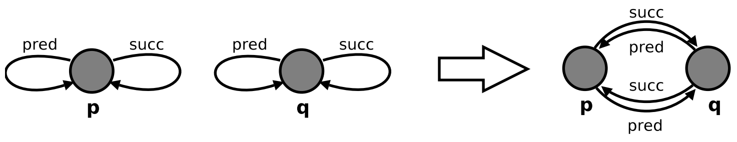


Figure 3.2: One peer joins the ring of another peer.

We start by having two single peers p and q. Each of them have their own ring, because they are their own successor. When one of the peers, say q, gets the remote reference of the other, q decides to join p's ring triggering the join event. Figure 3.2 shows the result of one peer joining the ring of the other. The result is the same if q joins p, or if p joins q.

To analyse the join algorithm, we will consider that the ring has at least two nodes. The algorithm works exactly the same for the case where there is only one node in the ring, with the difference that the predecessor and the successor of the joining peer are the same node, as it is shown in Figure 3.2. As we mentioned before, the join algorithm of the relaxed ring consists of three steps involving two peers each. The steps are depicted in Figure 3.3, where node q joins in between peers p and r. First steps connects the joining peer with its successor. Second steps connects the joining peer with its predecessor. Third step acknowledges the new peer between predecessor and successor.

Previous to the join process, peer q needs to find its successor candidate. In the case of the example it is r because $q \in]p, r]$. The first step of the join process is that r accepts q as predecessor, and notifying it about p. This is done with messages *join* and *joinOk*. After *joinOk*, peer q also gets to know that its responsibility is $]p, q]$, and r changes its responsibility to $]q, r]$. When this happens, peer q is already part of ring, because it is in the same ring as its successor r, and it has also defined its responsibility, although, it does not have a connection to its predecessor yet. At this point, we can say that q is living in a branch. Peer p still considers r as its successor, therefore, peer r keeps a reference to p in its *predlist*.

The second step of the join process occurs when peer q gets in contact with its predecessor, peer p. At that point, peer p changes its successor pointer to q, and it triggers the third step, where peer p notifies r about the change. The second step closes the ring, because q is not in a branch anymore. The third step improves the accuracy of the routing table. Note that responsibilities are not affected on the second step or on the third step, meaning that the address space was already consistent with the first step. If the second step never happens because q cannot talk to p, then, q will continue working in a branch, having r as root of its branch.

Algorithm 2 gives a detailed explanation of the first step of the join process. Step 1 is divided into two messages. To follow the example of Figure 3.3, we will call the joining peer q, and the direct neighbours as p and r. The *join* message sent from the joining peer q to the successor candidate r is only successful if new peer's identifier belongs to the responsibility of the contacted peer. Otherwise, the join request is routed to the real responsible as we will explain in the following section. Before the responsibility check is done, it is necessary for r to verify if its successor has not been suspected of having crashed.. This check on successor's fault-state is introduced to respect the invariant that every peer is in the same ring as its successor. It is not shown in the algorithm to avoid being verbose, and it is more related to implementation, than to the semantic of the protocol.

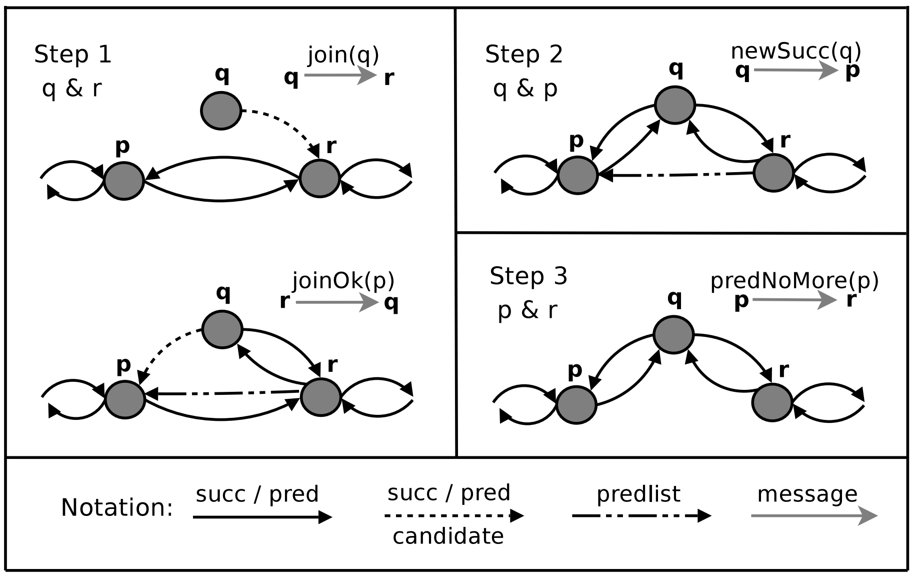


Figure 3.3: The join algorithm in three steps: contact the successor, contact the predecessor, and acknowledge the join.

A very important concept that differentiates the relaxed ring from other DHTs, is the predecessor list (*predlist*). We can observe that in Step 1, peer r stills remembers peer p through the predlist. This is important because if q cannot talk to p, peer p will still consider that r it is its successor, missing the peer in the middle.

Note that even when the *join* event has been triggered as result of the *replyLookup* sent from r to q, the verification of r's responsibility is still needed, because another node could have joined or failed within the range $]q, r]$, just before *join* was sent. In other words, it is possible that the responsibility of r has changed between the events *replyLookup* and *join*. In that case, q will be redirected to the corresponding peer with the route procedure, eventually converging to the responsible of its key. A related observation concerns the fact that in the algorithm we have excluded the right border of the range of responsibility. Excluding the right border we prevent peers from joining the ring with an identifier which is already in use.

The answer to a successful join is *joinOk*, which contains three important pieces of information: predecessor, successor and successor list. The successor is also used to confirm the correctness of the join, which could have been affected by simultaneous join events in the same address space, or the failure of some participants. As we already said, the predecessor helps q to already determine its responsibility range. This is different than all other ring networks we have studied. Finally, the successor list contains the needed resilient information, which is trimmed to size $log_k N$. The function trim(*list*, *size*) drops the last elements of *list* so that its length becomes equals to *size*. When q has finished the update of pointers, it notifies its predecessor p about itself as new successor, triggering the second step of this algorithm.

Algorithm 2 Join step 1 - new node contacts its successor

```
upon event ⟨ join | new ⟩ do
    if new ∈ ]pred, self[ then
        oldp := pred
        pred := new
        predlist := {new} ∪ {predlist}
        send ⟨ joinOk | oldp, self, succlist ⟩ to new
    else
        route ⟨ join | new ⟩
    end if
end

upon event ⟨ joinOk | p, s, sl ⟩ do
    if s ∈ ]self, succ[ then
        succ := s
        succlist := trim({s} ∪ sl, $log_k N$)
    end if
    if p ∈ ]pred, self[ then
        pred := p
        predlist := {p} ∪ predlist
        send ⟨ newSucc | self, succlist ⟩ to pred
    end if
end
```

The second step of the join algorithm involves peers p and q, closing the ring as in a regular ring topology. The step is described in Algorithm 3. The idea is that when p is notified about the join of q, it updates its successor pointer to q after verifying that is a correct join, and it updates its successor list with the new information. The successor list is propagated backwards with the message *updSucclist*, so as to notify the other peers in the ring about q. This resilient information will be discussed more in detail in Section 3.5. Functionally, this step is enough for closing the ring, but since we use a *predlist*, a third step is needed to acknowledge between the old direct neighbours the join of the new peer. The acknowledgement is done with message *predNoMore* sent from p to r, its old successor. If there is a communication problem between p and q, the event *newSucc* will never be triggered. In that case, the ring ends up having a branch, but it is still able to resolve queries concerning any key in the range $]p, r]$. This is because q has a valid successor and its responsibility is not shared with any other peer. It is important to remark the fact that branches are only introduced in case of non-transitive connectivity. If q can talk to p and r, the algorithm provides a perfect ring.

Algorithm 3 Join step 2 - Closing the ring

```
upon event ⟨ $newSucc$ | new, sl ⟩ do
    if new ∈ ]self, succ[ then
        send ⟨ $predNoMore$ | self ⟩ to $succ$
        succ := new
        succlist := trim({new} ∪ sl, $log_k N$)
        send ⟨ $updSucclist$ | self, succlist, $log_k N$ ⟩ to $pred$
    end if
end
```

In Step 3, peer p acknowledges its old successor r about the join of q. When *predNoMore* is triggered at peer r, this one can remove p from the *predlist*. The step is described in Algorithm 4.

Algorithm 4 Join step 3 - Acknowledging the join

```
upon event ⟨ $predNoMore$ | old ⟩ do
    if $old \in predlist$ then
        predlist := predlist \ {old}
    end if
end
```

No distinction is made concerning the special case of a ring consisting of only one node. In such a case, *succ* and *pred* will point to *self* and the algorithm works identically. The algorithm works with simultaneous joins, generating temporary or permanent branches, but never introducing inconsistencies. Failures are discussed in section 3.7. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

Theorem 3.3.1 *The relaxed ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.*

Proof 1 *Let us assume the contrary. There are two peers p and q responsible for the same key k, where $k > j$. To determine responsibility, it is not relevant if p and q have the same successor. If they have the same successor, each of them will forward lookup requests to the successor, and the successor will resolve the conflict. The problem appears when p and q have the same predecessor j. This means that $k \in]j, p]$ and $k \in]j, q]$ introducing an inconsistency because of the overlapping ranges. Then, we need to prove that the algorithm prevents two nodes from having the same predecessor. Consider the case where q joins between p and r. The join algorithm updates the predecessor pointer upon events* join *and* joinOk. *In the event* join, *r's predecessor is set to a new joining peer q. This means that no other peer was having q as predecessor because it is a new peer. Therefore, this update does not introduce any inconsistency. Upon event* joinOk*, the joining peer q establishes its responsibility having p as predecessor. Peer p was already a member of the ring. The only other peer that had p as predecessor before, it was peer r, being the peer that triggered the* joinOk *event. This message is sent only after r has updated its predecessor pointer to q, therefore, it modified its responsibility from $]p, r]$ to $]q, r]$, which does not overlap with q's responsibility $]p, q]$. Therefore, it is impossible that two peers has the same predecessor.*

3.4 Routing

We saw in the previous section that in the case of non-transitive connectivity, Step 2 of the join algorithm does not work, and therefore, branches are crated containing some of the peers of the relaxed ring. All peers that do not live in a branch are said to be at the core ring. Routing messages along the core ring works exactly as in Chord and DKS, forwarding to the closest preceding node. When a peer identifies that its successor is the candidate for being the responsible of the key, instead of answering immediately as in Chord, it forwards the request to its successor, setting a flag to indicate that it should be the *last* forwarding hop. If the successor is the real responsible according to its local responsibility, it will answer the lookup request. If it is not, meaning that it is the root of a branch, it backwards the lookup request using the *predlist*, choosing the closest succeeding peer. The flag *last* is kept to true, to keep the request being sent backwards, preventing cycles. The *lookup* event in Algorithm 5 sends the message *replyLookup* to the source that requested the lookup, and it routes the message otherwise.

The original source of the lookup request is kept when the message is transfered to the next peer. When the request reaches the responsible, it establishes a direct communication with the original source. This is what is called recursive routing. There are some implementation details related to fault tolerance and non-transitive connectivity that are not shown in the algorithm to avoid verbosity, but that are important for the implementation. While the message is being routed, it can happen that some peer in the path crashes. To prevent a message lost, the underlaying communication layer must provide reliable message sending. If the message does not reach the next peer, the message must be resend using a different finger. When the responsible is reached, it can happen that it cannot talk to the originator of the request. To solve this issue, there are a couple of strategies that can be used, for instance, follow the path back to the originator, taking into account that the path could be broken. Another possibility is to route the message using the normal routing algorithm as if it was a new message. A simpler way, proposed by [FLRS05], is to take a random finger and delegate the *replyLookup* to it. If that peer cannot talk to the original source either, then, it repeats the procedure of taking a random peer until some one can talk to the requesting peer.

Algorithm 5 Routing the lookup request

```
upon event ⟨ lookup | src, key, last ⟩ do
    if key ∈ ]pred, self] then
        send ⟨ replyLookup | self ⟩ to src
    else
        route ⟨ lookup | src, key, last ⟩
    end if
end

procedure route($msg$ | $\{args\}, dest, last$) is
    if last then
        p := getClosestSucceeding(predlist, dest)
        send ⟨ msg | {args}, dest, last ⟩ to p
    else if dest ∈ ]self, succ] then
        send ⟨ msg | {args}, dest, true ⟩ to succ
    else
        p := getClosestPreceeding(fingers, dest)
        send ⟨ msg | {args}, dest, last ⟩ to p
    end if
end procedure
```

3.5 Resilient Information

During the bootstrap and join algorithms we have mentioned *predlist* and *succlist* for resilient and routing purposes. The basic failure recovery mechanism is triggered by a peer when it detects the failure of its successor. When this happens, the peer will contact the members of the *succlist* successively. Apart from routing messages, the *predlist* can also be used to recover from failures when there is no predecessor that triggers the recovery mechanism. This is expected to happen only when the tail of a branch has crashed. There is no propagation of the *predlist*, because each peer builds its own list according to the behaviour of its direct neighbours.

Algorithm 6 describes how the update of the successor list is propagated while the list contains new information. The predecessor list is updated only during the join algorithm and upon failure recoveries. Note that the successor list is propagated to all peers in the predecessor list. Every time the list is propagated, the new peer is one place closer to the end of the list. When counter c reaches 0, it means that the new peer is already out of the successor list, and therefore, there is no need to continue propagating, because the list presents no new changes.

Algorithm 6 Update of successor list

```
upon event ⟨ $updSucclist$ | s, sl, c ⟩ do
    if $(s == succ) \wedge (c > 0)$ then
        succlist := trim({s} $\cup$ sl, $log_k N$)
        for all p in predlist do
            send ⟨ $updSucclist$ | self, succlist, c-1 ⟩ to $p$
        end
    end if
end
```

3.6 Leave Algorithm

There is no algorithm for handling graceful leaves. Any protocol designed to handle these kinds of leave events will have to deal anyway with partial failure during the leave. By handling failures in a general way, handling leaves comes for free. We consider graceful leave a special case of a failure. Some leave-messages can be sent to speed up "leave detection", instead of waiting for the timeout of the failure detector. Instead of having join/leave/fail algorithms, the relaxed ring is designed with join/fail algorithms.

3.7 Failure Recovery

To provide a robust system that can be used on the Internet, it is unrealistic to assume a fault-free environment or *perfect failure detectors*. A perfect failure detector is *strongly complete* and *strongly accurate*. Strongly complete means that all crashed nodes are eventually detected by all correct nodes. Strongly accurate means that a correct node is never suspected of failure. We assume that every faulty peer will be detected by all correct peers, therefore, it is possible to provide strong completeness. We also assume that a broken or very slow link of communication does not imply that the other peer has crashed, meaning that we can have false suspicions on correct peers. When the link recovers, we will eventually be able to communicate with the suspected peer, meaning that the failure detector will eventually be accurate. An *eventually perfect failure detector*, being strong complete and eventually accurate, is feasible to implement in clusters, local networks and the Internet. This is because a crashed node can always be suspected independently of the quality of the link used to communicate with it. Therefore, strong completeness is not difficult to achieve. If both nodes are alive, and messages sent between them are constantly repeated until they are acknowledged, any message will be eventually received, and therefore, any false suspicion of failure will be eventually corrected. That behaviour provides eventual accuracy. Therefore, both properties needed to build an eventually perfect failure detector are met, so this is the kind of detection we rely upon in our algorithms.

When the failure detector suspects that a peer has crashed, it triggers the *crash* event to the upper layer, being handled by the ring maintenance as it is described in Algorithm 7. The suspected node is removed from the resilient sets *succlist* and *predlist*, and added to a *crashed* set. If the suspected peer is the successor, a new successor is chosen from the successor list. The function *getFirst* returns the peer with the first key found clockwise. Since suspected peers are immediately removed from *succlist*, the new *succ* taken from the list is not suspected at the current stage. If the peer has just failed, the correspondent crashed event will be handled as one of the next operations. The message *fix* is sent to the new successor to inform it about the failure recovery. It is highly possible that the new successor has also detected the crash of the same peer, but it appears as predecessor to it. When the *pred* is suspected, if the peer belongs to the core ring, or it is in the middle of a branch, no action is taken. This is because it is the task of predecessor's *pred* to contact the new successor. Furthermore, the *predlist* of a core peer becomes empty when its *pred* is suspected, so it does not have any resilient information to take any action. If *predlist* is not empty, it is highly probable that the suspected peer did not have any predecessor. To avoid unavailability, we have decided that the current peer can find a new predecessor taking the last one clockwise. If there was a predecessor for the suspected peer, the correspondent *fix* message will eventually correct the inaccuracy. A safer action would be that the current peer never take any action if its *pred* is suspected, but that will create some unavailable ranges in some cases.

Algorithm 7 Failure recovery

```
upon event ⟨ crash | p ⟩ do
    succlist := succlist \ {p}
    predlist := predlist \ {p}
    crashed := {p} ∪ crashed
    if p == succ then
        succ := getFirst(succlist)
        send ⟨ fix | self, succ ⟩ to succ
    end if
    if (p == pred) ∧ (predlist ≠ ⊥) then
        pred := getLast(predlist)
    end if
end
```

In the case of a false suspicion, the event *alive* will be triggered by the failure detector when communication with the suspected peer is reestablished. There are three important cases to consider: the falsely suspected peer was the predecessor, the successor, or any other peer. In all cases, the peer is removed from the *crashed* set. If it is the predecessor, the *pred* pointer is corrected, and the peer is added back to the *predlist*. If it is the successor, the *succ* peer is corrected, the peer is added back to *succlist*, and in addition, the current peer sends the message *fix* to the successor, to run the correct protocol in case the successor has also falsely suspected the current peer. These actions are described in Algorithm 8.

Algorithm 8 Correcting a false suspicion

```
upon event ⟨ alive | p ⟩ do
    crashed := crashed \ {p}
    if p ∈ ]pred, self] then
        pred := p
        predlist := {p} ∪ predlist
    end if
    if p ∈ ]self, succ] then
        succ := p
        succlist := {p} ∪ succlist
        send ⟨ fix | self, succ ⟩ to succ
    end if
end
```

3.7.1 The Fix Message

The most common failure recovery occurs at the core ring, or in the middle of a branch, where the successor of the crashed peer does not have any other peer in the *predlist*. In other words, the successor of the crashed peer is not the root of a branch. In such case, if a peer *p* detects that its predecessor *pred* has crashed, it will not send any message. It is *pred*'s predecessor who will contact *p*. Figure 3.4 shows the recovery mechanism triggered by a peer when it detects that its successor has a failure. The figure depicts two equivalent situations. Figure 3.4(a) corresponds to a regular crash of a node in the core ring. The situation at Figure 3.4(b) shows a crash in the middle of a branch, which is equivalent to the situation at the core ring as long as there is a predecessor that detects the failure.

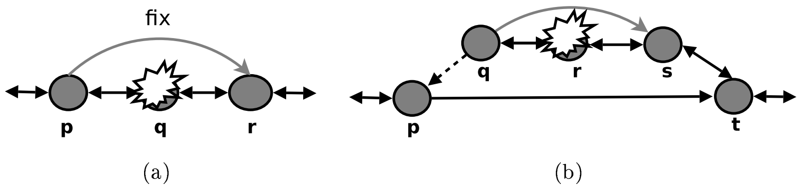


Figure 3.4: Failures simple to handle: (a) Peers p and r detect q has crashed. Peer p triggers the recovery mechanism. (b) In a branch, q and s detect that r has crashed. Only q triggers failure recovery.

The basic principle of the failure recovery is that whenever a peer suspects that its successor has crashed, it sends the *fix* message to the first peer it finds in its successor list. Algorithm 9 shows how the *fix* message is handle. Normally, a peer receives this message from a peer *p* after suspecting that its predecessor has failed, but it is possible that it receives it while its predecessor is still alive due to false suspicions. The first thing to check is whether *pred* is suspected or not. If such is the case, peer *p* becomes the new *pred*, and it is added to the *predlist*. If *pred* is alive, it is possible that *p* is a better predecessor than the current predecessor. This situation can occur when the root of a branch has crashed. In such case, two peers will send the *fix* message to the same successor candidate. Depending on the order, peer *p* will be a better predecessor of the current *pred* or not. We will discuss this situation more in detail in Section 3.7.3. If *pred* is alive, and *p* is not a better *pred*, the message is routed to the correspondent peer. The extra parameter *s* is used to indicate the successor candidate of the peer *p*. The first time the event is handled, *s* matches the current peer *self*. In that case, peer *p* is added to *predlist* because *p* considers the current peer as its current successor.

Algorithm 9 Successor of crashed node receives fix message

```
upon event ⟨ $fix$ | p, s ⟩ do
    if pred $\in$ crashed then
        pred := new
        predlist := {p} $\cup$ {predlist}
        send ⟨ $fixOk$ | self, succlist ⟩ to $p$
    else if p $\in$ ]pred, self] then
        pred := p
        predlist := {p} $\cup$ {predlist}
        send ⟨ $fixOk$ | self, succlist ⟩ to $p$
    else
        if self == s then
            predlist := {p} $\cup$ {predlist}
        end if
        route ⟨ $fix$ | p, s ⟩
    end if
end
```

When the *fix* event is accepted, the predecessor of the crashed peer receives the message *fixOk*. This message can come directly from its successor candidate, meaning that the fix occurred at the core ring, or in the middle of a branch. The only action to be taken at this stage is to propagate the new successor list. If the *fixOk* message comes from a different peer, with a key better than the current successor, it is necessary to change the *succ* pointer, and notify the old successor candidate about the change with message *predNoMore*, as it was shown in Section 3.3, Algorithms 3 and 4. Propagation of *succlist* happens anyway.

Algorithm 10 Fix message accepted

```
upon event ⟨ $fixOk$ | s, sl ⟩ do
    if s ∈ ]self, succ[ then
        send ⟨ $predNoMore$ | self ⟩ to $succ$
        succ := s
    end if
    succlist := trim({s} ∪ sl, $log_k N$)
    send ⟨ $updSucclist$ | self, succlist, $log_k N$ ⟩ to $pred$
end
```

3.7.2 Join and Failure

Knowing the recovery mechanism of the relaxed ring, let us come back to our joining example and check what happens in cases of failures. In our example, peer q joins in between peers p and r. If q crashes after sending message *join* to r, peer r still has p in its *predlist* for recovery. If q crashes after sending *newSucc* to p, p still has r in its *succlist* for recovery. If p crashes before event *newSucc*, p's predecessor will contact r for recovery, and r will inform this peer about q. If r crashes before *newSucc*, peers p and q will contact simultaneously r's successor for recovery. If q arrives first, everything is in order with respect to the ranges. If p arrives first, there will be two responsible peers for the range $]p, q]$, but one of them, q, is not known by any other peer in the network, and it fact, it does not have a successor, and then, it does not belong to the ring. Then, no inconsistency is introduced in any case of failure during a join event. In case of a network partition, these peers will get divided in two or three groups depending on the partition. In such case, they will continue with the recovery algorithm in their own rings. Global consistency is impossible to achieve, but every ring will be consistent in itself.

Figure 3.5 shows two simultaneous crashes together with a new peer s joining as predecessor of the peer used for recovery t. If the recovery *fix* message sent from p arrives first, the ring will be fixed before the new peer joins, resulting in a regular join of s in between p and t. If the new peer starts the first step of joining before the recovery, it will introduce a temporary branch because of its impossibility of contacting the faulty predecessor r. When the *fix* message arrives, peer t will route the message to s establishing the connection between p and s, fixing the ring. If p and s cannot talk, the branch will remain permanent.

3.7.3 Non Trivial Suspicions

There are failures more difficult to handle than the ones we have already analysed. Figure 3.6 depicts a broken link between peers p and q. This means that p suspects q and tries to fix the ring contacting peer r, the first one on its *succlist*. Peer q also suspects p, but it waits until p's predecessor sends the *fix*

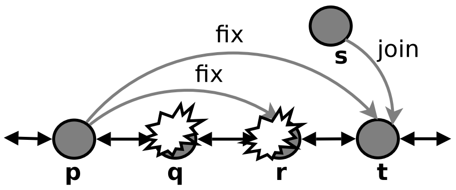


Figure 3.5: Multiple failure recovery and simultaneous join. Peer p detects the crash of its successor q. First successor candidate r has also crashed. Peer p contacts t at the same time peer s tries to join the network.

message. Peer q does not have any other choice than wait, because its *predlist* is empty once p was removed from it. Peer r does not accepts p's fix message because q is still alive. Peer r routes the *fix* message to q, which is not able to talk to p, so the recovery ends up over there. As result, peer q moves to a branch, and peer p is added to r's *predlist*, being a valid configuration of the relaxed ring.

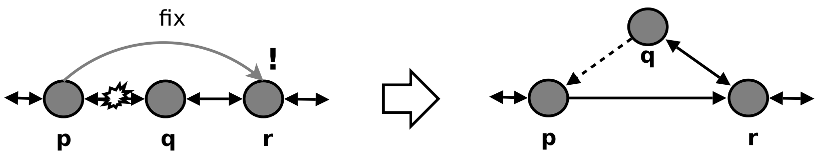


Figure 3.6: Broken link generating a false suspicion.

Figure 3.7 shows the crash of the tail of a branch. In such case, there is no predecessor to trigger the recovery mechanism. Depending on how the branch was created, it is possible that peer r has peer p in its *predlist*. That would be the case if peer q arrived after peer r. If peer p is in r's *predlist*, peer r would be able to modify its *pred* pointer filling in the gap of range $]p, q]$. If peer r arrived in between q and s, then, there is no way that r knows about p as a valid predecessor. In such case, peer r will not take any action, and the gap in the range $]p, q]$ will remain until churn gets rid of it. If the tail of the branch, q, has not really failed but it has a broken link with its successor r, then, it becomes temporary isolated and unreachable to the rest of the network.

Figure 3.8 depicts the failure of the root of a branch. The main difference with previously analysed crashes is that in this case, there are two recovery messages triggered. In the example, peer r crashes, and peers p and q send the fix message to peer t. If message from peer q arrives first to peer t, the algorithm handles the situation without problems. If message from peer p arrives first, the branch will be temporary isolated, introducing a temporary inconsistency on the range $]p, q]$. This limitation of the relaxed ring is well defined in the following theorem.

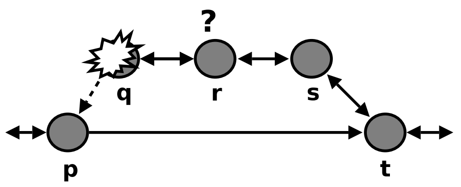


Figure 3.7: Failure of the tail of branch, nobody triggers failure recovery.

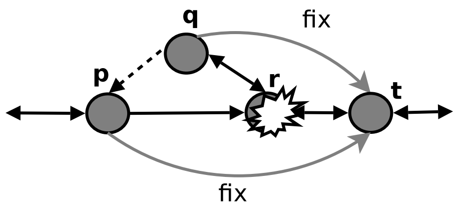


Figure 3.8: The failure of the root of a branch triggers two recovery events

Theorem 3.7.1 *Let r be the root of a branch, t its successor, q its predecessor, and* predlist *the set of peers having r as successor. Let p be any peer in the set, so that $\{p, q\} \in predlist$. Then, when peer r crashes, peers p and q will try to contact t for recovery. A temporary inconsistent lookup may be introduced if p contacts t before q contacts t. The inconsistency will involve the range]p, q], and it will be corrected as soon as q contacts t for recovery.*

Proof 2 *There are only two possible cases. First,* pred *contacts* succ *before p does it. In that case,* succ *will consider* pred *as its predecessor. When p contacts* succ*, it will redirect it to* pred *without introducing inconsistency. The second possible case is that p contacts* succ *first. At this stage, the range of responsibility of* succ *is $]p, succ]$, and of* pred *is $]p', pred]$, where $p' \in [p, pred]$. This implies that* succ *and* pred *are responsible for the range $]p', pred]$, where in the worse case $p' = p$. As soon as* pred *contacts* succ *it will become the predecessor because $pred > p$, and the inconsistency will disappear.*

Theorem 3.7.1 clearly states the limitation of branches in the systems, helping developers to identify the scenarios needing special failure recovery mechanisms. Unfortunately, the problem seems difficult to prevent without loosing the flexibility of the relaxed ring. An advantage of the relaxed ring topology is that the issue is well defined and possible to detect, because in perfect rings peers have *predlist* with one element maximum. Therefore, if a peer has more than one peer of its *predlist* it means that it is the root of a branch. Identifying and understanding this situation, improves the guarantees provided by the system to build fault-tolerant applications on top of it.

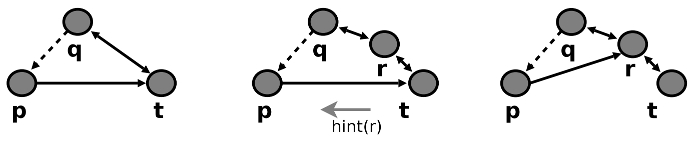


Figure 3.9: Pruning branch with *hint* message.

3.8 Pruning Branches

The first results measuring the size of branches had good average values, but the worse cases influenced too much the routing efficient. That result led the search for pruning branches without risking lookup consistency, which is what we describe in this section. Let us consider a configuration where peer q is in a branch because it could not establish communication with its predecessor p. Both peers, p and q, have peer t as successor, being t the root of the branch. The situation is depicted at Figure 3.9. From that point, a new peer r joins in between q and t. With the current join algorithm, the branch will increase its size even if p can talk to r. Because of non-transitive communication, if p can not talk to q, it does not mean that p can not talk to r. An improvement on the join algorithm will be that peer t sends a *hint* message to node p about peers joining as t's predecessor. If p can contact the *hinted* node r, it will add it as its successor, making the branch shorter. It is necessary that peer r adds p to its *predlist*, and that p notifies t so that t removes p from its *predlist*. All these changes can be done with messages *fix* and *predNoMore* as we already described previously. Note that modification will not modify the predecessor pointers of r or s. Therefore, lookup consistency is not compromised. Figure 3.9 also depicts how peer p updates its successor pointer preventing the branch from going unnecessarily. The algorithm we have just described in words is presented in Algorithm 11.

Algorithm 11 Fix message accepted

```
upon event ⟨ hint | s ⟩ do
    if s ∈ ]self, succ[ then
        send ⟨ predNoMore | self ⟩ to succ
        succ := s
        send ⟨ fix | self, succ ⟩ to succ
    end if
end
```

To trigger the *hint* message, a first naive approach would be to send the message to all peers in the *predlist* as soon as *joinOk* was sent to the joining node. However, this approach can generate some overlapping of successor pointers making some parts of the ring unavailable. Let us analyse the situation using Figure 3.10. At the left of the figure we observe that peer s joins the ring as predecessor of t, t being the root of a branch. After s is accepted as part of the ring, the other peers in t's predecessor list are peers p and q. In our naive approach, t would send the *hint* message to p and q. The upper most configuration at the right side shows a bad configuration as result of these two messages. It happens if peer p can contact s, but peer q cannot. There is no lookup inconsistency introduced, however, whenever p receives a request for a key k in range $]p, q]$, it will forward it to peer s, who will send it to r, who will try to send it to q, but the connection is broken, making peer q unavailable in whatever path that goes through peer p. A good configuration of the naive approach can be seen on the second configuration at the right side of the figure. Such configuration results when peers p and q can talk to s. This is actually the best possible configuration to reduce the size of the branch. It is, nevertheless, risky because of the reasons we just explained.

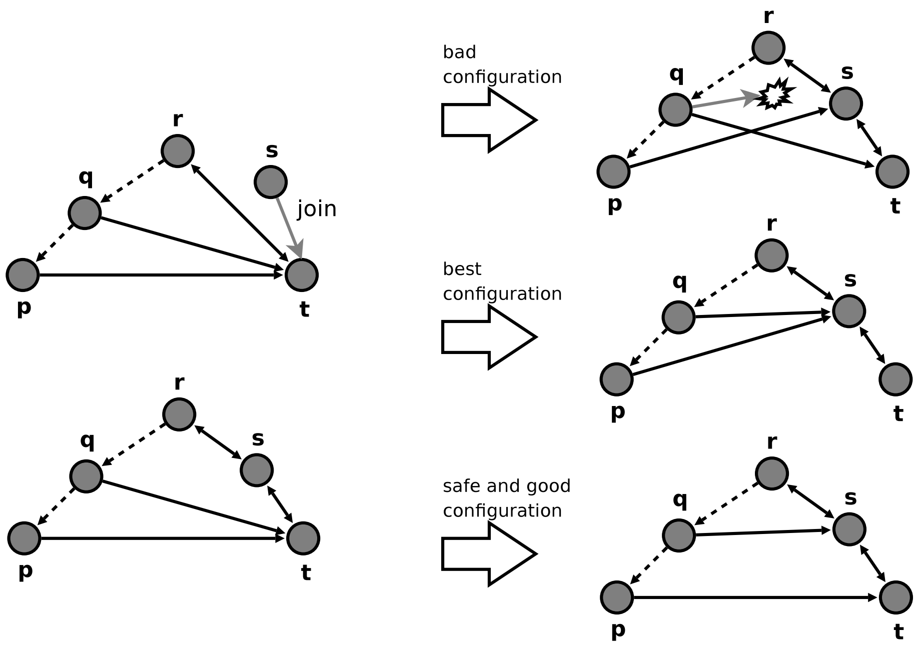


Figure 3.10: Different configurations pruning branches.

A safer approach is that t sends the *hint* message only to the closer peer anti-clockwise, in this case, peer q. Like that, if peer q cannot talk to s, no unavailability is introduced. However, peer p would never be notified about peer s, preventing an improvement in the topology. A more sophisticated approach would be to implement an acknowledgement to the *hint* message, so that peer t can continue sending *hint* messages to other peers in its *predlist*. It can be implemented by sending *hint* to the rest of the peers only when *predNoMore* arrives to the root of the branch. Note that this part of the algorithm only improves the topology, but does not change the correctness of it.

3.9 Adaptable Routing-Table Construction

In this section, we propose a hybrid reconfiguration mechanism for the routing table called PALTA*: a Peer-to-peer AdaptabLe Topology for Ambient intelligence*. It was originally conceived for an ambient intelligent scenario, but it can be used on any kind of network. This algorithm takes advantage of the best features of a fully connected network when the number of peers is small enough to allow peers to manage this kind of topology. When the network becomes too large to maintain a fully connected topology, the algorithm will automatically adapt the network configuration to become a relaxed ring, which can handle a large number of peers by executing more complex algorithms for self-managing the distributed network. We consider different aspects concerning the transition between networks: adaptation of the base algorithms, maintaining the network's coherence and self-healing from inefficient configurations. Evaluation follows in Chapter 6.

PALTA finger table will change its organization when the network size reaches a defined threshold. We will refer to this limit as ω. Even when during the discussion of this chapter we consider the value of ω as uniform for all nodes, the algorithm is designed such that every node can define its own ω to adapt its behaviour according to its own capacities.

To successfully implement this dynamic schema, we need to analyze how the topology will evolve when peers join or leave the network. When the network is created and the number of peers is below ω, the joining peers will perform the *fully connected* algorithm which simply creates a full mesh of peers. When peers detect a network size above ω, all the incoming joining requests will be handled using *relaxed ring*'s join algorithm. The same methodology is followed when, after a number of disconnections, the network becomes smaller than ω. In such case, peers will change their joining algorithm from relaxed ring to fully connected. The general idea is depicted in Figure 3.11. At the left side we observe a fully connected network that has reached its $\omega = 6$. When the seventh peer joins the network, it will just connect to $log(N)$ fingers instead of to all of them.

To be able to make the transition from a fully connected network to a ring, peers need to precisely identify their successors and predecessors at any time. Algorithm 12 shows that the *join* event in PALTA can be seen almost as a method dispatcher, with the subtlety that it checks its predecessor and predecessor pointers in every join before triggering the *join* event of the FULL module. In case that ω is already reached, it is the algorithm of the relaxed ring who will take care of *pred* and *succ* accordingly. For this section, we have divided the algorithms into three modules: FULL, RING and PALTA.

The value of ω can change right after sending message *join* and-or *joinOk*.

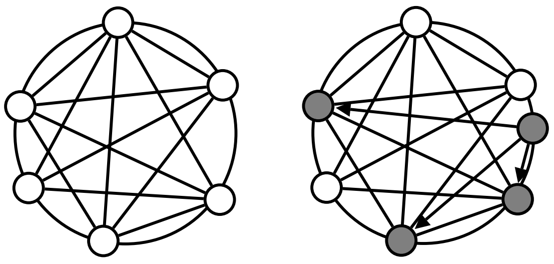


Figure 3.11: Adaptive routing table with PALTA.

Algorithm 12 Join for PALTA: Adapted fully connected algorithm with transition to relaxed ring

```
upon event ⟨ join | new ⟩ do
    if size(peers) < $\omega$ then
        checkSuccPred(new)
        trigger ⟨ FULL.join | new ⟩
    else
        trigger ⟨ RING.join | new ⟩
    end if
end

procedure checkSuccPred(id) is
    if id $\in$ ]self, succ[ then
        succ := id
    end if
    if id $\in$ ]pred, self[ then
        pred := id
    end if
end procedure
```

Due to that, there is no way of knowing if a reply message *joinOk* will correspond to the fully connected topology or to the relaxed ring. Peers need to adapt dynamically to this situation whenever needed. Algorithm 13 shows how this event is overloaded in PALTA. The first case corresponds to *joinOk* as in the ring, with information of the predecessor, successor and successor list. If the routing table is higher than ω, the event is delegated to the relaxed ring module. If we are in a small network, predecessor and successor are accordingly checked, and the successor list is used to trigger the fully connected algorithm, which will be used until reaching ω.

Algorithm 13 Join for PALTA: Overloaded event *joinOk*

```
module PALTA

upon event ⟨ joinOk | p, s, sl ⟩ do
    if size(peers) $< \omega$ then
        checkSuccPred(new)
        trigger ⟨ FULL.joinOk | s, sl ⟩
    else
        trigger ⟨ RING.joinOk | p, s, sl ⟩
    end if
end

upon event ⟨ joinOk | src, srcPeers ⟩ do
    if size(peers) $< \omega$ then
        checkSuccPred(new)
        succList := succList $\cup$ srcPeers
        trigger ⟨ FULL.joinOk | src, srcPeers ⟩
    end if
end
```

3.10 Conclusion

In this chapter we have presented the relaxed ring topology for fault-tolerant and self-organizing peer-to-peer networks. The topology is derived from the simplification of the join algorithm requiring the synchronisation of only two peers at each stage. As a result, the algorithm introduces branches to the ring. These branches can only be observed in presence of connectivity problems between peers, and they help the system to work in realistic scenarios. The complexity of the lookup algorithm is a bit degraded with the introduction of branches. However, we will analyse the real impact of this degradation in Chapter 6, and we will see that it is almost negligible. In any case, we consider this issue a small drawback in comparison to the gain in fault tolerance and cost-efficiency in ring maintenance.

The topology makes feasible the integration of peers with very poor connectivity. Having a connection to a successor is sufficient to be part of the network. Leaving the network can be done instantaneously without having to follow a departure protocol, because the failure-recovery mechanism will deal with the missing node. The guarantees and limitations of the system are clearly identified and formally stated providing helpful indications to build fault-tolerant applications on top of this structured overlay network.

This chapter was also dedicated to describe our self-adaptable finger table called PALTA. It is based on the existing fully connected and relaxed ring topologies, with adaptations to make them work together. This hybrid topology features self-organizing and self-adapting mechanisms to ensure a complete connectivity among the connected peers and take advantage of the current network state to have a better use of the available resources. It benefits from fully connected small networks and it makes a smooth transition to larger ones, being able to scale as a large-scale structured peer-to-peer network.

Chapter 4

The Relaxed Ring in a Feedback Loop

For the strength of the pack is the wolf,
and the strength of the wolf is the pack.

"The Wolf" - Rudyard Kipling

In previous chapters, we have discussed about self-managing behaviour of peer-to-peer networks, specially self-organization and self-healing. In this chapter, we focus the discussion around self-management using feedback loops to analyse self-* properties of the relaxed ring. The chapter provides a different design approach that targets two goals: identify patterns of self-managing behaviour in the relaxed ring, and second, use those patterns to design other Beernet's components. Feedback-loops, as we will see in the next section, are present in every self-managing system, and therefore, applying them in software design seems to be a reasonable path to achieve self-management.

The feedback loop can be seen as a design tool that complements the relax approach. It helps in the design and analysis of self-managing systems in general, and by applying them to the relaxed ring, it provides a different view to understand its algorithms. In case the reader is not interested in feedback loops, it is possible to skip this chapter because the algorithms of the relaxed ring have been already presented in Chapter 3.

4.1 Background

Taken from system theory, *feedback loops* can be observed not only in existing automated systems, but also in self-managing systems in nature. Several examples of this can be found in [Van06, Bul09], where feedback loops are introduced

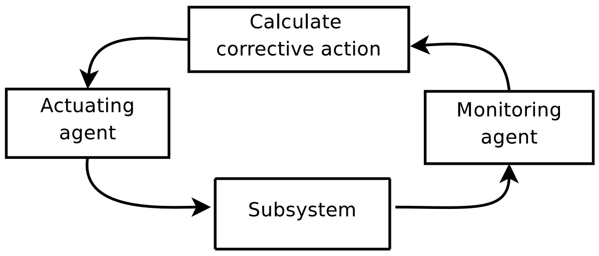


Figure 4.1: Basic structure of a feedback loop ([Van06]).

as a designing model for self-managing software. The loop consists out of three main concurrent components interacting with the subsystem. There is at least one agent in charge of monitoring the subsystem, passing the monitored information to a another component in charge of deciding a corrective action if needed. An actuating agent is used to perform this action in the subsystem. Figure 4.1 depicts the interaction of these three concurrent components in a feedback loop. These three components together with the subsystem form the entire system. It has similar properties to *PID controllers*, which is one of the basic units of feedback control in industrial control systems. A PID controller works on a process variable to keep it as close as possible to a desirable *setpoint*. At every loop, the controller measures the process variable and calculates the difference with the setpoint. If an adjustment is needed, the controller modifies the inputs of the process. The feedback loops presented in this chapter are an equivalent in software to PID controllers in automated systems.

The goal of the feedback loop is to keep a global property of the system stable. In the simplest case, this property is represented by the value of a parameter. For instance, in an air-conditioning system, the parameter is the temperature of the room. This parameter is constantly monitored. When a perturbation is detected, a corrective action is triggered. A *negative feedback will make the system react in the opposite direction of the perturbation.* Positive feedback increases the perturbation. For instance, if an increase in room's temperature is detected, running the air-conditioning is an action to decrease the temperature, and therefore, it creates negative feedback. Turning on the heating system will increase room's temperature, going in the same direction as the perturbation, and therefore, producing positive feedback.

Coming back to the example of the air-conditioning, we can see the *room* where the system is installed as the subsystem. A *thermometer* constantly monitors the temperature in the room giving this information to a *thermostat*. The thermostat is the component in charge of computing the correcting action. If the monitored temperature is higher than the wished temperature, the thermostat will decide to *run the air-conditioning* to cool it down. That action corresponds to the actuating agent.

Since every component executes concurrently, the model fits very well for modelling distributed systems. There are many alternatives for implementing every component and the way they interact. They can represent active objects, actors, functions, etc. Depending on the chosen paradigm, the communication between components can be done, for instance, by message passing or event-based communication. The communication may also be triggered by pushing or pulling, resulting in eager or lazy execution.

Independent of the strategy used for communication, it is important to consider asynchronous communication as the default when distributed systems are being modelled.

4.1.1 Other Feedback Loops

In this section we give a brief overview of different approaches to design self-managing systems. As we will see, all of them have a common basic principle: *monitor* the subsystem, *decide* on an action, *execute* the action, and monitor again. By monitoring again, the loop is restarted.

Automatic elements are defined in Automatic Systems [KC03] as *control loops*, also known as MAPE-loops. The word MAPE come from *monitor*, *analyze*, *plan* and *execute*. Figure 4.2 describes the architecture of a control loop, which is very similar to the basic feedback loop we have shown in Figure 4.1. Both loops need to monitor the subsystem and execute a plan on it. One difference is that the autonomic element includes the monitor and the actuator inside the autonomic manager. The feedback loop consider those parts of the system as completely independent components. Another difference is that the component in charge of taken the decision is split in two components, one for the analysis, and the other one for the planning. In the feedback loop, those two actions are considered as being part of calculating a corrective action. We can say that the MAPE-loop gives a finer granularity of behaviour associated to each component, but the feedback loops is more modular because monitors and actuators are independent.

Another interesting component of the MAPE-loop is the *knowledge*, which is used by all other four components. In the feedback loop, the knowledge is implicitly present in the main component that decides on the corrections. However, that knowledge cannot be used directly by the monitor or the actuator, because there is no shared state between the components to keep them as independent as possible.

In 1970, the SRI's Sharkey robot used the SPA software architecture to develop autonomic behaviour. The architecture, which is depicted in Figure 4.3, was extremely simple and followed same principle we have discussed in this chapter: *sense* de environment, *plan* an action, and *act*. Once the action was performed, the robot sensed the environment again to repeat the process. The problem with this architecture was that the robot was not able to plan quick enough, therefore, when it performed the action the environment had already changed, making the plan invalid.

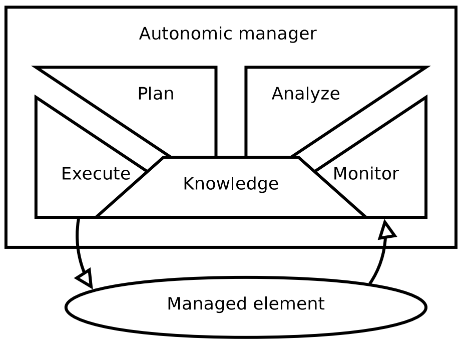


Figure 4.2: Autonomic element using MAPE-loop [KC03].

The problem with the response time of the robot using the SPA architecture motivated other architectures having different layers, each one of them with its own SPA loop. Each layer had a different task to performed, improving the response time of the global system. That implies that feedback loops can interact with each other, and that each one of them must perform a specific task within a larger system. It is very important that all interaction between feedback loops is well defined, because not always the combination of them will provided the desired behaviour.

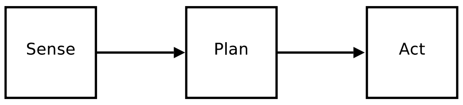


Figure 4.3: SPA software architecture used in Robotics in 1970.

The interaction between feedback loops can occur in layers as in robotic systems, or as in the three-layer architecture [Gat97, KM07], where there is a *hierarchy* of feedback structures, where one loop can manage another one directly. Another way of interacting is through *stigmergy*. One loop modifies a subsystem which is also monitored by another loop. This means that both loops share a common resource. The modification done by one loop will take effect on a new action from the other loop, affecting the same subsystem again, influencing the actions of the former loop. When two or more loops communicate indirectly through the subsystem they monitor and effect, we say that they communicate through stigmergy.

4.2 Join Algorithm

In this section we describe the same join algorithm from Chapter 3, but now using feedback loops. The algorithm was not originally designed using such technique, but thanks to its self-managing behaviour we were able to extract the feedback loop pattern in the same way other designed patterns are discovered in other systems. We conceive the whole peer-to-peer network as self-managing system, where the network is the subsystem we want to monitor. We want to keep its functionality despite the changes that can occur on the network, being basically join and fail events, and temporary or permanent broken links. The structure of the ring is the global property that needs to be kept stable. New peers joining, and current peers leaving or failing represent perturbations to the ring structure. Therefore, these events must be monitored. Broken links are very difficult to distinguish from crashed peers, and therefore, a broken link and a crashed peer will be monitored as the same event, without affecting the semantic of the protocols. If a broken link is fixed, it will appear as a false suspicion of a failure, as it will be explained in Section 4.3

Messages sent during the process of joining, and the update of the predecessor and successor pointers are shown in Figure 3.3. In the example, node q wants to join the network having r as successor candidate. Peer r is a good candidate because it is the responsible for key q. Node q send a join request to r. Whereas the *join* event triggered by peer q is a perturbation, event *joinOk* is a correcting action because it fixes the successor-predecessor relationship between peers q and r.

After *joinOk* is triggered, a branch is created. Then, a second correcting action is needed to entirely close the ring. This action is represented by the event *newSucc* sent from peer q to p. The monitoring agents are in charge of detecting perturbations in the network. Correcting actuators can be seen as three different actions: update routing table (successor and predecessor), trigger event (correcting ones) and forward request (in case a peer wants to join at the wrong place). The routing table does not only include predecessor and successor. It also includes fingers for efficient routing and resilient sets for failure recovery. It is also part of the *knowledge* of the computing component.

To analyse the *join* feedback loop more in detail, we will split it into three loops: one to accept the new predecessor, one to notify the former predecessor about its new successor, and one the acknowledge the join between the former direct neighbours. Figure 4.4 depicts the feedback loop that handles a new predecessor. The event to monitor is the *join* message. If the message is sent to the wrong successor candidate, the request is forwarded using the shared resource amount loops, which is the network. When the *join* message reaches the correct successor, the successor sends the message *joinOk* to its new predecessor. Every node keeps on monitoring for new *join* messages, including the new peer, the one that forwarded the request, and the one that handled it. The *joinOk* message is monitored by a different feedback loop. There is no actuator included in Figure 4.4 that modifies the value of the predecessor because, as we already mentioned, *pred* and *succ* pointers are inside the computing component as part of its knowledge. Therefore, the action is only a change in the knowledge of the component, where that knowledge does not need to be monitored. We will see in Section 4.6 that this is not always the case.

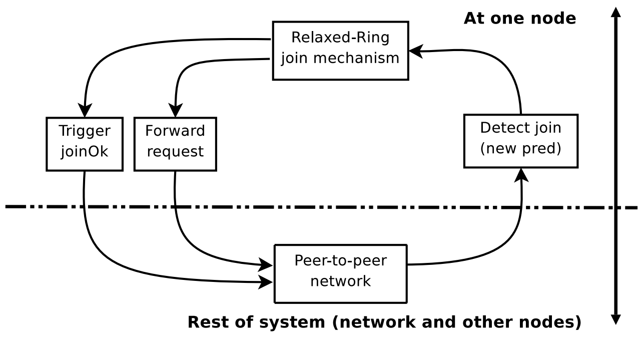


Figure 4.4: Feedback Loop to handle a new predecessor.

After the new joining peer has triggered a perturbation on the peer-to-peer network by sending the *join* message, it monitors the network waiting for the *joinOk* message sent by its successor. The *joinOk* message will contain information about the predecessor of the joining peer, and therefore, the next action to be taken in the loop is to notify that predecessor about the current peer as being its new successor. This is done by sending the *newSucc* message. If the message is never received, the peer remains in a branch. This feedback loop can be seen in Figure 4.5. We can observe that dividing each task into an orthogonal but collaborative loop, makes the loops very straightforward, and therefore, simpler to analyse and debug.

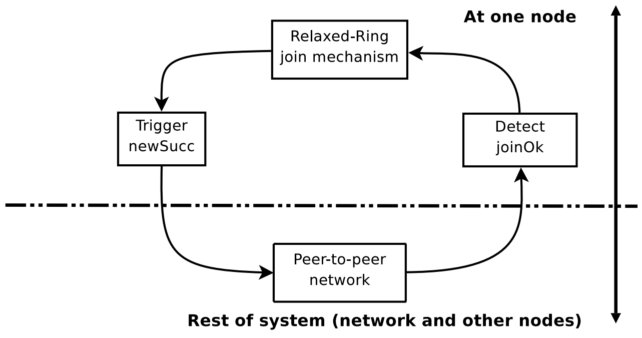

At one node
Relaxed-Ring join mechanism
Trigger newSucc
Detect joinOk
Peer-to-peer network
Rest of system (network and other nodes)

Figure 4.5: Feedback loop for join accepted.

The only remaining action to integrate the new peer in the ring is the notification of the former predecessor to its old successor about the new joining peer. This is done with message *predNoMore* upon detection of message *newSucc*. We can observe that each of this three loops interact through the network using it as a share resource. The acknowledgement loop can be seen in Figure 4.6.

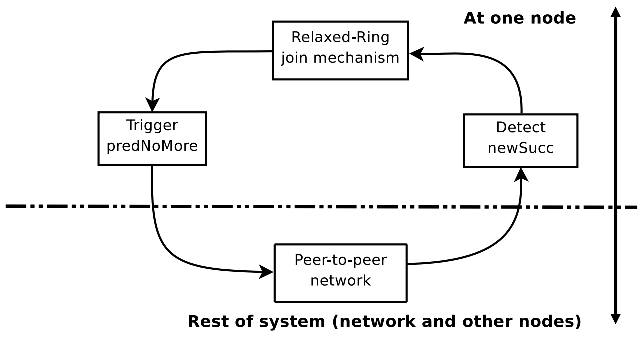

At one node
Relaxed-Ring join mechanism
Trigger predNoMore
Detect newSucc
Peer-to-peer network
Rest of system (network and other nodes)

Figure 4.6: Feedback Loop to handle a new successor.

Every peer monitors the network independently, and the correcting actions performing the ring maintenance are running concurrently in every peer. Despite the fact that each loop has its own goal, they all share a local common resource at each peer, to which they must have exclusive access: the successor, predecessor and predecessor list. Because they all share the same local resource, it is more convenient to have them all implemented in the same component. Figure 4.7 shows the full join loop, describing what are all the messages being monitored, and what are the possible actions that the loop can take. Every event triggered by a peer is monitored by the destination peer, unless there is a failure in the communication. In that case, a *crash* event will be triggered and treated by the failure recovery mechanism, which is also a feedback loop as we will see in the following section. Interleaving of events do not introduce inconsistencies in the order of the peers in the ring, as it was shown in Section 3.3.

4.3 Failure Recovery

As we described in Chapter 3, instead of designing a costly protocol for peers gently leaving the network, leaving peers are treated as peers having a failure. Like this, solving problem of failure recovery will also solve the issue of leaving the network. The feedback loops presented in this section were also extracted with post-design pattern mining.

Observing the relaxed ring as a self-managing system, we identify that the crash of a peer also introduces perturbations to the structure of the ring. Therefore, crashes must be monitored. To provide a realistic solution, *perfect failure*

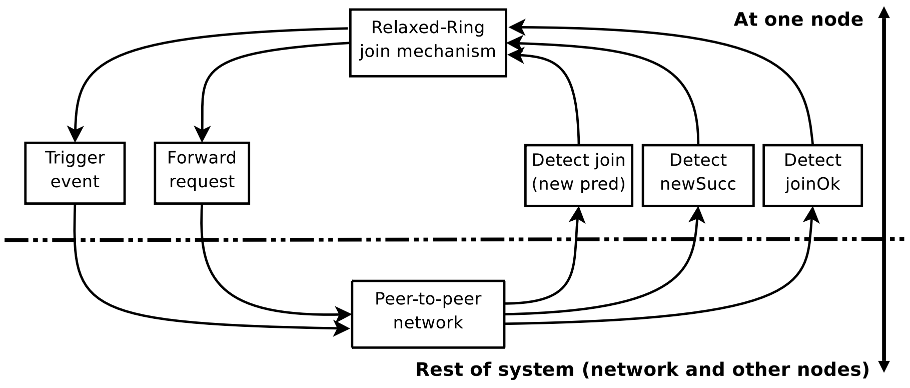


Figure 4.7: Join algorithm as a feedback loop.

detectors cannot be assumed as the monitoring agent. Perfect failure detectors and the reason for not using them were described in Section 3.7. If there is a suspicion, an accurate detector guarantees that the suspected process has really failed. In reality, broken links and nodes with slow network connection are very often, generating a considerable amount of false suspicions. Several kinds of network address translation (NAT) devices also introduce problems with failure detection. NAT devices are very popular at Internet home users. This implies that the failure recovery mechanism can only rely on *eventually perfect* failure detector, meaning strongly complete and *eventually* accurate. Being eventually accurate means that the detector can falsely suspect a correct peer, but that eventually the suspicion will be corrected. That correction is done by triggering the $alive(p)$ event to indicate that a peer p is really alive, and that it was falsely suspected. Because of this, not only $crash(p)$ events must be monitored, but also $alive(p)$ events. When these two events are detected, the network must update routing tables and trigger correcting events.

In the relaxed ring architecture we use the *fix* message as correcting agent for stabilising the relaxed ring. If the network become stable, the *fixOk* event will be monitored. This negative feedback can be observed in Figure 4.8. When the recovery mechanism is mixed with some concurrent *join* events, it is possible that the successor candidate chosen by the recovery mechanism is not the best one for the peer. In such case, and as described in detail in Chapter 3, the *fix* message will be forwarded to the real successor of the peer, which will contact the recovering peer with *fixOk*. In such case, the recovering peer will notify its first successor candidate with the message *predNoMore*, as in regular ring maintenance. The situation can also occur when the root of a branch crashes, triggering two simultaneous *fix* message on the same peer.

The interaction between feedback loops is an interesting issue to analyse because big systems are expected to be designed as a combination of several loops.

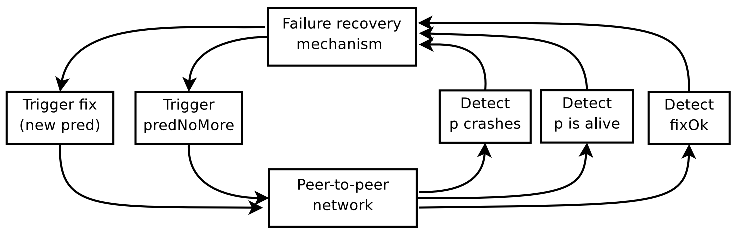


Figure 4.8: Failure recovery as a feedback loop.

Let us consider a particular section of the ring having peers p, q and r connected through successor and predecessors pointers. Figure 4.9 describes how the ring is perturbed and stabilised in the presence of a failure of peer q. Only relevant monitored and actuating actions are included in the figure to avoid a bigger and verbose diagram. In the figure we have added the events *newPred* and *predOk*. These events do not exist explicitly in the algorithms. They represent the concept behind analog events in ring-maintenance and failure-recovery. The events *join* and *fix* are triggered when a peer needs a successor, and therefore, it wants to become the new predecessor (*newPred*) of its successor candidate. In the case of *join*, the candidate is chosen because it appears to be the responsible for the its key. In the case of *fix*, it is because it is the successor of a crashed peer. The other analogy comes with events *joinOk* and *fixOk*, which actually represent that the *newPred* request was accepted *predOk*. In the case of *joinOk*, the event is triggered when the range of local responsibility includes the joining peer. The *fixOk* message is sent after testing not only the responsibility, but also the failure status of the current predecessor.

Given the explanation of how the feedback loop on Figure 4.9 was made, we proceed with the explanation when peer q crashes in between peers p and r. Initially, the crash of peer q is detected by peers p and r (1). Both peers will update their routing tables removing q from the set of valid peers. Since p is q's predecessor, only p will trigger the correcting event *fix*, trying to become the *newPred* (2). Because the new predecessor can be verified with the key-range and the failure status, we observe that it affects the two feedback loops, being a communication between them. The *newPred* event will be monitored by peer r (3), testing the message against the conditions defined by the ring maintenance, and the failure recovery mechanism. The correcting action *predOk* will be sent to p (4), together with the corresponding update of the routing table. Then, the event *predOk* will be monitored (5) by the failure recovery component to perform the correspondent update of the routing table. Since the *predOk* event is also detected by the join loop, both loops will consider the network stable again.

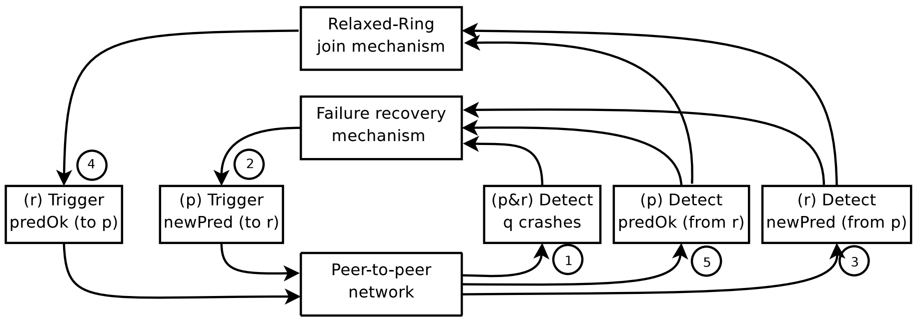

Relaxed-Ring join mechanism
Failure recovery mechanism
(r) Trigger predOk (to p)
(p) Trigger newPred (to r)
(p&r) Detect q crashes
(p) Detect predOk (from r)
(r) Detect newPred (from p)
Peer-to-peer network
1
2
3
4
5

Figure 4.9: Peers p and r detect failure of q, fixing the ring with an interaction of feedback loops.

4.4 Failure Detector

We already saw how to recover from the failure of a peer, and how to update the routing table when we detect a false suspicion. We study now the implementation of a self-tuning failure detector. First of all, we describe a generic implementation of an eventually perfect failure detector, and then, we analyse how can we tune some of its parameters using three feedback loops with different goals. After deducing the feedback loop pattern from the ring-maintenance algorithms, this failure detector was originally designed using feedback loops.

We can find the design and implementation of several failure detectors in Chapter 2 of [GR06]. All of them are based on the principle of the *heartbeat*. The idea is that each process sends periodically a HEARTBEAT message to all processes it is connected to. Since every process does the same, each one of them receives heartbeats from the others. The algorithm for eventually perfect failure detection works as follows. A process p sends a heartbeat to all processes it knows about. Let us call that set of processes Π. After the heartbeat is sent, p launches a timer, which corresponds to the timeout for waiting heartbeats from each process in Π. This timer also represents the period of time to initiate the next heartbeat round. When a heartbeat is received from another process q, q is added to the *alive* set. When the timeout is triggered, The set $suspected = (\Pi \setminus alive)$ represents the processes that are suspected of having failed. The correspondent $crash(i)$ event is triggered to the upper layer for all process $i \in suspected$. Then, a new heartbeat round begins. Process p sends again a heartbeat message to everybody and starts collecting heartbeats from its connections. If a heartbeat coming from a process i arrives, where $i \in suspected$, it means there is a false suspicion. This will be detected at the next timeout when both sets *alive* and *suspected* will contain process i in their sets. When this situation occurs, the corresponding event $alive(j)$ is triggered for all $j \in (alive \cap suspected)$. If there is no such j, it means that the timeout could be shorter to detect crashed peers quicker. Therefore, and a new timeout is chosen by reducing Δ from it. If there is such j, meaning that the failure detector is not accurate enough, the new timeout will be increased by Δ trying to prevent the false suspicions to happen again. The timeout will be incremented until there is no intersection between sets *alive* and *suspected*.

Figure 4.10 shows the feedback loop that allows the eventually perfect failure detector to tune the period of time representing the timeout and the heartbeat interval, which is actually the same. The loop is very simple. The failure detector monitors if there is any false suspicion at the end of every heartbeat round. If there are, the timeout is incremented until reducing the amount of false suspicions. This means that it is negative feedback. If there are no false suspicions, the timeout is decremented to speed up the detection.

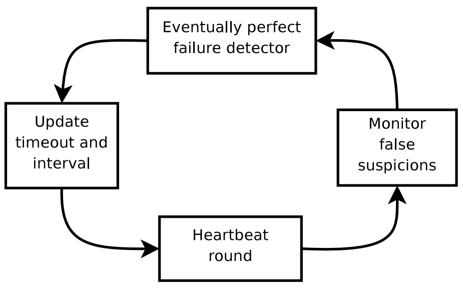

Eventually perfect failure detector
Update timeout and interval
Monitor false suspicions
Heartbeat round

Figure 4.10: Feedback loop improving timeout and heartbeat interval of an eventually perfect failure detector.

The algorithm works correctly according to the promised guarantees, but it can be largely improved. First of all, let us identify the problems of it. The fist issue is that the heartbeat interval is the same as the timeout, therefore, the values cannot be independently adapted. The heartbeat interval should be adapted due to bandwidth consumption policies, and the timeout should be adapted with respect to the round-trip time. In addition, the round-trip time is very often different for each connection with other processes. Therefore, the failure detector should have one timeout per connection, and not one global for all processes. Another problem with this algorithm is that all detections goes at the speed of the slowest connection. Let us understand why. Consider processes a, b and c, where a is connected to b and b is connected to c. The connection between a and b is very slow, and between b and c is very fast. Because the connection between a and b is slow, in the first round b will receive normal heartbeat from c but it will falsely suspect that a has crashed. When b receives a's heartbeat after the timeout, it will increase its own timeout, meaning that it will increase the period of time for sending its own heartbeat. Process b will continue increasing its period until a is no longer falsely suspected, meaning that the timeout is adapted to the slowest connection. Because b sends heartbeats less often to c, process c will start falsely suspecting that b is crashing, and therefore, it will also adapt its period of interval to not suspect b. That means that c adapts its heartbeat interval to the speed of connectivity between a and b, which is very slow. This adaptation will be easily propagated to the rest of the network which will adapt its interval to the slowest connection between two processes.

Following the previous analysis, we will improve the failure detector by using two different values, one for the heartbeat interval and another one for the timeout. To adapt the timeout according to the round-trip time of the communication links, we will use a *ping-pong* protocol instead of a simple heartbeat. The idea is that each *ping* message must be eagerly acknowledged with the correspondent *pong*. That will allows us to measure the round-trip time of the link. That will imply that we would be able to adapt a different timeout for each connection, and that failures will be detected a quicker. Having more independent parameters, the failure detector can be tuned having different goals in mind as it is shown in the feedback loop of Figure 4.11. We will consider three feedback loops sharing a common resource, which is the values of the ping-interval and each timeout. Note that instead of a heartbeat round, now we have a ping-pong round to monitor. First of all, the *Bureau of Accuracy* has the goal of reducing the amount of false suspicions, therefore, if there is a false suspicion detected it will increase the value of the timeout. This is negative feedback, and it is equivalent to the loop we described for the original eventually perfect failure detector, with the difference that it does not affect the ping-interval, only the timeout. The *Ring Manager*, in charge of keeping the routing tables as accurate as possible, is interested in having failure detection as quick as possible. Therefore, if the dead-state ratio is too large at the end of the ping-interval, it means that the timeout is too large, because too many peers crashed while the timer was still running. That means that it will decrease the value of the timeout to decrease the dead-state ratio. This is also a negative feedback, but it modifies the value of the timeout in the opposite direction as the *Bureau of Accuracy*. That will imply that these two modification will help each timeout to converge to the ideal value.

The *Ring Manager* will also modify the ping-interval to be more frequent so as to detect failures quicker. This change will imply a larger traffic, which probably will contradict the goal of the *Bandwidth Supervisor* who wants to keep the bandwidth consumption as close as possible to the defined policies, to be more efficient in the use of resources. As soon as the bandwidth consumption is higher that the expected cost, the *Bandwidth Supervisor* will modify the ping-interval to send *ping* messages less often. The result would be a reduction in the consumption, meaning that it is a negative feedback. Both loops modify the ping-interval in opposite directions, tuning the system to find the optimal value that respect all policies as good as possible. If policies are really conflicting, then, the values will continue oscillating around the best possible solution. In this failure detector, we can see how different feedback loops with different goals can collaborate to build a self-tuning system.

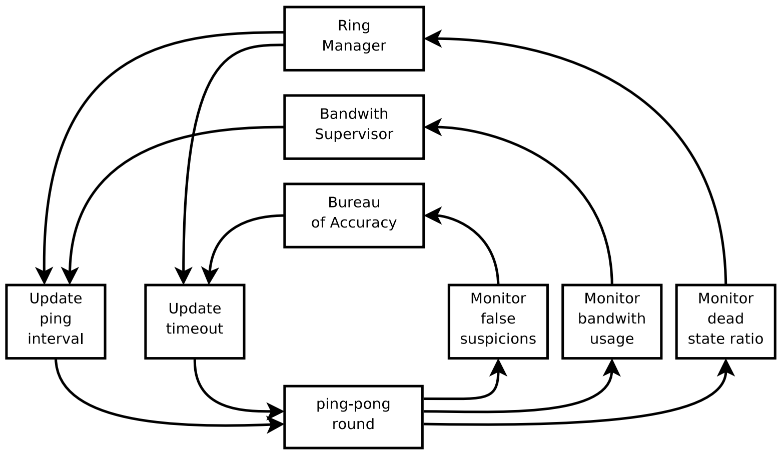


Figure 4.11: Self-tuning failure detector designed with a set of feedback loops with conflicting goals.

4.5 Finger Table

We can also observe a feedback loop pattern in the management of the finger table of the relaxed ring. The failure detector we study in Section 4.4 is one of the monitors used by the finger table maintenance manager, as it is the monitor of many other feedback loops we have presented in this chapter. Whenever a finger is suspected of having failed, a lookup for the correspondent key is triggered as a corrective action. The goal is to find a new responsible for the key to become the new finger. The loop is closed when the *lookupReply* is received. This loop follows the *correction-on-change* strategy. To complement this strategy, we also use *correction-on-use*, which means that every information of other peers going though the current peer is monitored. As soon as a better finger is detected the finger table is updated. When the finger table is updated, some acknowledgement messages are sent through the network. The feedback loop is shown in Figure 4.12.

Many other ring-based peer-to-peer networks use periodic stabilization to maintain the finger table as accurate as possible. The idea is to periodically ask every finger for its predecessor. If it happens to be a better finger than the current one, then, the finger is updated. If the finger does not respond to the periodic request, a lookup is triggered to replace the finger. The strategy is also a feedback loop that corrects the subsystem periodically.

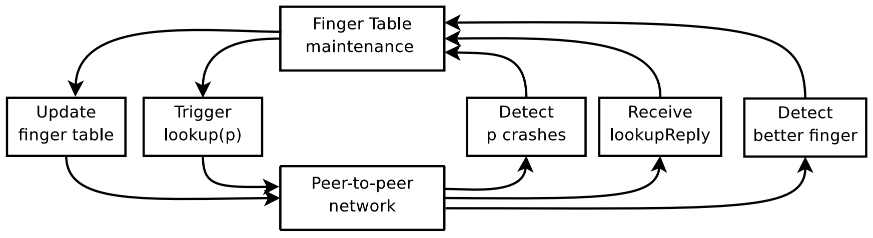


Figure 4.12: Maintenance of the Finger Table.

4.6 Self-Adaptable Routing-Table

Once we have identified the feedback loop pattern on the way the finger table is constructed, we use feedback loops to design a self-adaptable routing table, called PALTA, which was already described in Section 3.9. PALTA's feedback loop is shown in Figure 4.13. The monitors, actuators and the component that decides the corrective actions are placed at every node. The monitored subsystems correspond to the whole peer-to-peer network, and the routing table. The last one is also placed at the node.

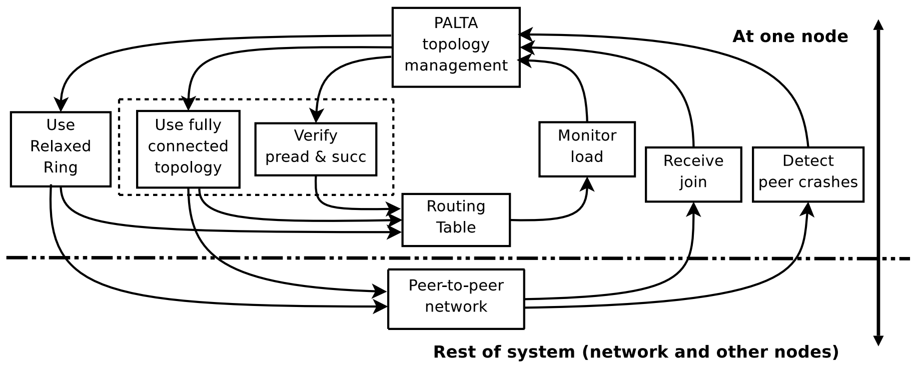


Figure 4.13: Self-Adaptable topology as a feedback loop.

As explained in Section 3.9, when a new node wants to enter the network, it sends a *join* message to its successor candidate. This message is sent through the network. Since every node is monitoring the network, the *join* message will be received by the PALTA component. PALTA is also monitoring the load of the routing table. This information is used to decide how to react to the *join* message. If the load is below ω, PALTA will use the fully connected mechanism together with its own verification of the predecessor and successor. Both actions are used to update the routing table, modifying its load, which will be monitored once again, as in every loop. The fully connected mechanism will also trigger some messages in the peer-to-peer network to modify its state. If the load of the routing table has already passed the ω threshold, PALTA will use the relaxed ring joining mechanism, which will also update the routing table and trigger some messages for the involved nodes.

We can observe some similarity between PALTA's feedback loop and the acclimatized room briefly described in Section 4.1. The thermostat in the room will use the heating system or the air-conditioner depending on whether the temperature is below or above the desired goal. PALTA decides its actuators according to load of the routing table with respect to ω. In the acclimatized room, the temperature is measure periodically, being triggered by a timer. In PALTA, loop's monitoring process is triggered when the *join* message is detected by its receiver.

The loop also monitors failures of peers triggering the corresponding failure recovery mechanism. This mechanism is chosen by PALTA according to the load of the routing table, as it is done with the joining process. This is coherent with what is explained in Section 3.9. All other messages related to the joining process and the failure recovery, such as *joinOk*, *newSucc*, are also present as monitoring event, but they have been omitted from Figure 4.13 for legibility.

4.7 Conclusion

Decentralised systems in the form of peer-to-peer networks present many advantages over the classical client-server architecture. However, the complexity of a decentralised system is higher due to the lack of a central point of control. Therefore, it requires self-management. In this chapter we show how feedback loops, taken from existing self-managing systems, can be applied in the design of a peer-to-peer networks. Using feedback loops, we can observe that the system is able to monitor and correct itself, keeping the relaxed ring structure stable despite the changes caused by regular network operations and node failures.

We have also shown how feedback loops are combined using the subsystem as a way of interacting from one loop to the other. The eventually perfect failure detector is an interesting case study to show how loops having conflicting goals can be combined to build a self-tuning system. The failure detector itself is then used to build other feedback loops, where its role is to monitor the failure state of other peers. The self-adaptable behaviour of the PALTA finger table becomes more accessible by modelling it as a feedback loop. It is clear that the ω value is constantly monitored to adapt the behaviour of the routing table whenever the threshold is reached. It is also clearer how actuators are chosen dynamically according to the values that are monitored.

Chapter 5

Transactional DHT

Nothing's what it seems to be, I'm a replica.

"Replica" - Sonata Arctica

The basic operations provided by a DHT are `put(key, value)` and `get(key)`. We have seen in Chapter 2 that this is not enough to provide fault tolerance, and that a replication strategy should be used to guarantee data storage. Replicas are not simple to maintain independently of the chosen replication strategy. Therefore, it is very convenient to add transactional support to the DHT, so that replica management becomes encapsulated and atomic operations over a set of items is guaranteed.

The two-phase commit protocol (2PC) is one of the most popular choices for implementing distributed transactions, being used since the 1980s. Unfortunately, its use on peer-to-peer networks is very inefficient because it relies on the survival of the transaction manager, as explained further in section 5.1. A three-phase commit protocol (3PC) has been designed to overcome the limitation of 2PC. However, 3PC introduces an extra round-trip which results in higher latency and increased message load. We will see how transactional support based on Paxos consensus [MH07, GL06] works well in decentralized systems. This algorithm is especially adapted for the requirements of a DHT and can survive a crash of the coordinator during a transaction. Compared to 3PC, it reduces latency and overall message load by requiring less message round-trips.

We extend the Paxos consensus algorithm with an eager locking mechanism to fit the requirements we identified in synchronous collaborative applications. A notification layer is also added to the transactional layer support, which can be used by any of the transactional protocols we will describe. In this chapter we also make an analysis of replica maintenance, and how it is related to the transactional layer.

One of the important contributions of this dissertation is the design of a lock-free transactional protocol for key/value-sets, extending the key/value pair data abstraction. Key/value-sets were also used in OpenDHT [RGK+05], but only eventual consistency was provided, and no support for transactions. Our protocols provide strong consistency when reading from the majority of the replicas, and eventual consistency when the set is read from a single replica.

5.1 Two-Phace Commit

The pseudo-code in Algorithm 14 implements a swap operation within a transaction. The objective is that the instructions from the beginning of the transaction (BOT) until its end (EOT) are executed atomically to avoid race conditions with other concurrent operations. The values of *item_i* and *item_j* are stored on different peers. The operators *put* and *get* are replaced by *read* and *write* to differentiate a regular DHT from a transactional DHT. Since the operations have different semantics, as we will see in section 5.8, it is justified to use different keywords.

Algorithm 14 Swap transaction

```
BOT
    x = read(item_i);
    y = read(item_j);
    write(item_j, x);
    write(item_i, y);
EOT
```

To guarantee atomic commit of a transaction on a decentralized storage, two-phase commit uses a *validation* phase and a *write* phase, coordinated by a *transaction manager* (TM). All peers responsible for the items involved in the transaction, as well as their replicas, become *transaction participants* (TP). Initially, the TM sends a request to every TP to *prepare* the transaction. If the item is available, the TP will lock it and acknowledge the *prepare* request. Otherwise, it will reply *abort*. The *write* phase follows *validation* once the replies are collected by the TM. If none of the participants voted *abort*, then the decision will be *commit*. When the participants receive the commit message from the TM, they will make the update permanent and release the lock on the item. An abort message will discard any update and release the item locks.

The problem with the 2PC protocol is that relies too much on the survival of the transaction manager. If the TM fails during the validation phase, it will block all the TPs that acknowledged the prepare message. A very reliable TM is required for this protocol, but it cannot be guaranteed on peer-to-peer networks. Figure 5.1 depicts 2PC protocol showing two possible executions. The diagrams do not include the client, but they concentrate on the interaction

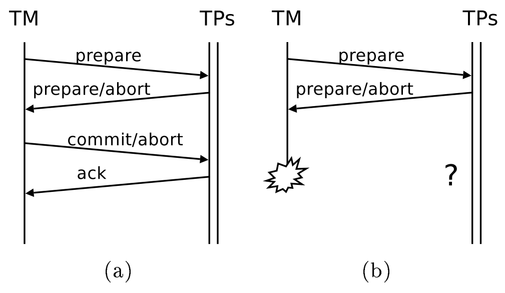


Figure 5.1: Two-Phase Commit protocol (a) reaching termination and (b) not knowing how to continue or unlock the replicas because of the failure of the transaction manager.

between the TM and the TPs. Figure 5.1(a) shows a successful execution of the protocol where the TPs get the confirmation of the TM about the result of the transaction. Figure 5.1(b) spots the main problem of this protocol. If the TM crashes after collecting the locks of the TPs, the TPs remained locked forever if the algorithm is *crash-stop*. PostgreSQL [Pos09], a well established object-relational database management system, implements 2PC as a *crash-recovery* algorithm, meaning that the TM can reboot and recover the state before the crash to continue with the protocol. Discussing with PostgreSQL developers, we have learned that a transaction could hang for a whole weekend before the locks are released again. This kind of behaviour is not feasible in peer-to-peer networks when there is no certainty that a peer that leaves the network will ever come back.

5.2 Paxos Consensus Algorithm

The 3PC protocol avoids the blocking problem of 2PC at the cost of an extra message round-trip and a timeout to release locks, which produces performance degradation. This solution might be acceptable for cluster-based applications but not for peer-to-peer networks, where it is better to have less rounds with more messages than adding extra rounds to the protocol. Having more messages increases the bandwidth consumption. Having more rounds increases the execution time of a protocol. A protocol with a longer execution time becomes more fragile because it increases the probability of having a partial failure. Therefore, it is more desirable to have more messages introduced in a protocol, than having more rounds of messages between the participants. This problem led to the recent introduction of [MH07] based on Paxos consensus [GL06].

The idea is to add replicated transaction managers (rTM) that can take over the responsibility of the TM in case of failure. The other advantage is that decisions can be made considering a majority of the participants reaching

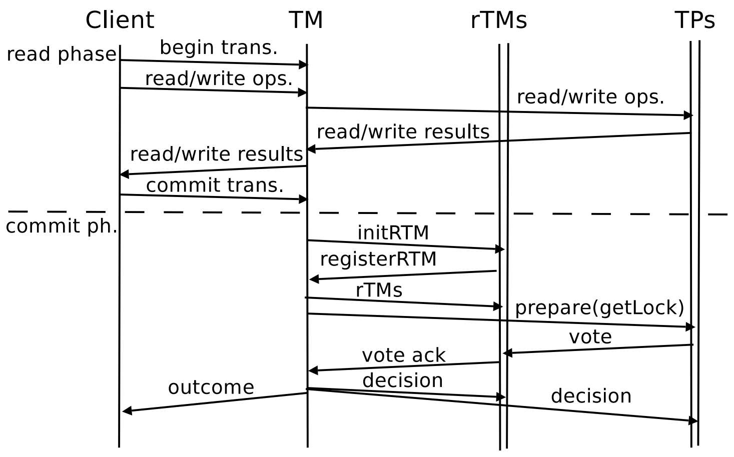


Figure 5.2: Paxos consensus atomic commit on a DHT.

consensus, and therefore, not all participants need to be alive or reachable to commit the transaction. This means that as long as the majority of participants survives, the algorithm terminates even in presence of failures of the TM and TPs, without blocking the involved items.

Figure 5.2 describes how the Paxos-consensus protocol works. The client, which is connected to a peer that is part of the network, triggers a transaction to read/write some items from the global store. When the transaction begins, the peer becomes the transaction manager (TM) for that particular transaction. The whole transaction is divided in two phases: *read phase* and *commit phase*. During the *read phase*, the TM contacts all transaction participants (TPs) for all the items involved in the transaction. TPs are chosen from the peers holding a replica of the items. The modification to the data is done optimistically without requesting any lock yet. Once all the read/write operations are done, and the client decides to commit the transaction, the *commit phase* is started.

To commit changes to the replicas, it is necessary to get the lock of the majority of TPs for all items. But, before requesting the locks, it is necessary to register a set of replicated transaction managers (rTMs) that are able to carry on the transaction in case that the TM crashes. The idea is to avoid locking TPs forever. Once the rTMs are registered, the TM sends a *prepare* message to all participants. This is equivalent to request the lock of the item. The TPs answer back with a *vote* to all TMs (arrow to TM removed for legibility). The vote is acknowledged by all rTMs to the leader TM. The TM will be able to take a decision if the majority of rTMs have enough information to take exactly the same decision. If the TM crashes at this point, another rTM can take over the transaction. The decision will be *commit* if the majority of TPs voted for commit. It will be *abort* otherwise. Once the decision is received by the TPs, locks are released.

The protocol provides atomic commit on all replicas with fault tolerance on the transaction manager and the participants. As long as the majority of TMs and TPs survives the process, the transaction will correctly finish. These are very strong properties that will allow the development of collaborative applications on a decentralized system without depending on a server.

5.2.1 Self Management

We can observe the property of self-configuration in this transactional protocol in the way the replicated transaction managers are chosen, and in the way the replicas are found. Even when replicas should not change from one transaction to the other, unless there is some churn, the set of TM and rTMs tends to be different in every transaction. There is no intervention in the election of the members of these sets, they just follow the high level policies and self-configure to run the transaction. The self-healing property can be observed when the TM fails. One of the rTM is elected to become the new TM, and it finishes the transaction. The election is done following the identifiers in the ring, so they all reach an agreement.

5.2.2 Non-Transitive Connectivity

We have explained that relaxed ring's algorithms do not rely on transitive connectivity, therefore, it tolerates the presence of peers behind NAT devices. Nevertheless, Paxos, and all Trappist's protocols, are described as any peer being able to contact any other peer participating in the transaction. This could be a problem if some peers are behind NAT devices and are unable to establish a direct communication between them. However, the communication between peers participating in a transactional protocol is entirely delegate to lower layers in charge of messaging. For efficient, Beernet will try to establish a direct channel between protocol participants, but if there is no success, the relaxed ring is used to route messages, guaranteeing that at the Trappist layer, any peer can talk to any other peer. More explanation about the component in charge of the direct sending of messages between peers is given in Section 7.6.

5.3 Paxos with Eager Locking

An implementation of Wikipedia on Scalaris [SSR08, PRS07] using Paxos consensus algorithm, shows that Paxos can be successfully used to build scalable systems with decentralized storage. We will also describe Sindaca, a community-driven recommendation system that will be presented in Section 8.1. Sindaca is another example of building decentralized systems using Paxos. These systems are designed to support asynchronous collaboration between application's users. The fact that Paxos consensus protocol works with optimistic locking fits well asynchronous collaboration. However, this locking strat-

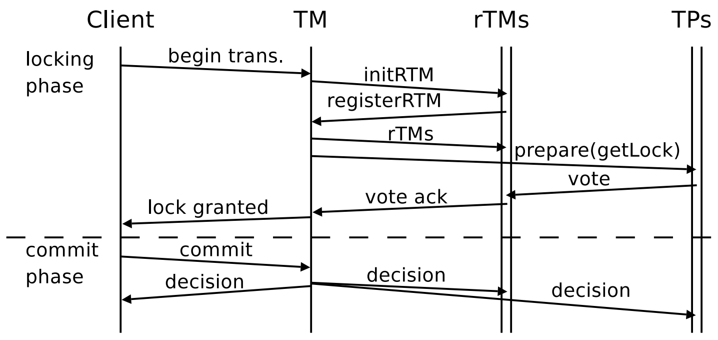


Figure 5.3: Paxos consensus with eager locking and notification to the readers.

egy limits the functionality of synchronous collaborative applications such as DeTransDraw, a collaborative drawing tool that we will describe in Section 8.2.

DeTransDraw has a shared drawing area where users actively make updates and observe the changes made by other users. If two users make modifications to the same object at the same time, at the end of the their work, when they decide to commit, only one of them will get her changes committed, and the other one will loose everything. Because users are working synchronously, the probability that this happens is much larger than in applications such as Wikipedia or Sindaca. This is why a pessimistic approach with eager locking is needed.

We have adapted Paxos to support eager locking adding a notification mechanism for the registered readers of every shared item. We have implemented this new protocol in Trappist, the transaction layer support of Beernet, with the possibility of dynamically choosing between the two Paxos protocols. Given this choice, the application can decide the protocol to be used depending on the functionality that is provided to the users.

Figure 5.3 depicts the adapted protocol with eager locking. The read-phase and commit-phase from the original protocol has been replaced by *locking-phase* and *commit-phase*. The read phase disappears because the transaction manager tries eagerly to get the relevant locks to proceed with the transaction. Once the locks are collected, the client is informed of the result. The goal is to prevent users from trying to start working on items that are already locked. The client of the transaction starts working on the changes on the items as soon as the transaction begins. Starting to work on an item is actually the trigger of the transaction.

When the user stops making modifications, it triggers the commit-phase. The transaction manager can take the decision immediately because the majority of the votes have been already collected at this stage. The decision is propagated to the client, the replicated transaction managers and transaction participants, as in the original Paxos algorithm. As there is no read-phase, it is important that the decision is transmitted to the TPs and rTMs together with the new state of the item, and not only a *commit/abort* message.

This protocol is unfortunately more fragile than Paxos without eager locking. By dividing the acquisition of locks and the decision of commit, we can propagate information to the readers more eagerly, but we increase the period of time where the TM and the client need to survive. If the client fails before committing its final value, the locks need to be retrieved and released. If there is a false suspicion, we could end up having two clients claiming the lock of an item. To solve this, it is necessary to add timestamps not only to the values of the items, but also to the locks. If the TM crashes before the client commits its final decision, one of the rTM becomes the new TM, and it needs to inform the client about the failure recovery. Once the client is notified about the recovery, the client can commit the transaction using the new TM.

Self-management The self-configuration property with respect to the set of rTMs is inherited from the previous Paxos protocol without eager locking. Self-healing is also achieved, because the system can recover from the crash of the client and the TM, and it can complete if the majority of participants survives. The mechanism is a bit more complex due to the split of the locking and commit phase.

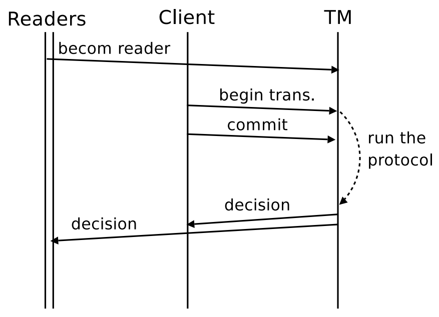
5.4 Notification Layer

The modification with eager locking provides notification to the readers every time an item is locked and updated. Sometimes it is not necessary to get a notification on locking, and only the update is important. In such case, it is interesting to have a layer of notification independent of the protocol used to update the item. This kind of feature is useful to implement applications such an online score board, where only a few peers modify the state of the application, and many peers participate as readers. For the readers, it is not necessary to get a notification that some value is currently being updated. They just need to get the last value of the item.

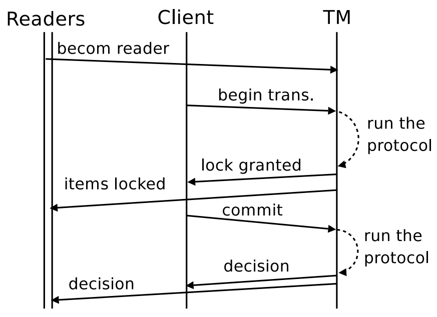
The layer consists of a reliable multicast that sends a notification to all subscribed readers of an item. To make the multicast efficient, if the amount of readers is smaller than $log(N)$, a direct message sending can be performed. If the amount is larger, the update message can be transmitted using the multicast layer of the peer-to-peer network.

5.5 Lock-Free Key/Value-Sets

Data collections are a useful data abstraction as we will see in Chapter 8. Providing strong consistency can be achieved using any of the above described protocols, but that would imply locking the collection every time a modification is performed. Following our design philosophy, by relaxing some conditions we



(a)



(b)

Figure 5.4: Notification layer protocol. Peers register to each item by *becoming readers*. Figure (a) shows one notification for the decision if the transaction is run with Paxos. Figure(b) shows notifications for locking and decision, if transaction is run with Eager Paxos.

are able to get rid of locks, allowing concurrent modifications to the collection, and still providing strong consistency. We relax versioning and the order of the elements in the collection. The result is the key/value-set abstraction as a complement to key/value-pairs. OpenDHT [RGK$^+$05] also uses key/value-sets, but they only provide strong consistency nor transactional support.

As an example of the usefulness of unordered value-sets, we can refer to the data of a collaborative drawing tool consists mainly of description of figures and the canvas. Each figure can be represented as a key/value pair storing all properties of the figure. The description of the canvas is just a set of figures being drawn on it. If the canvas is represented as a key/value pair, only one user would be able to add or remove a figure from the canvas at the time. This locking restriction is completely unnecessary because there is no order on the figures. Their position is stored on each one of them, not in the canvas. We will explain more about such a collaborative drawing tool when we explain DeTransDraw in Section 8.2.

Another example is a recommendation system where users can express their preferences. We have built a prototype of such system named Sindaca, which we will describe in detail in Section 8.1. If videos are the items being recommended, we have conceived a decentralized data-base where the information about the video is stored in a key/value pair, with a value-set associated to it to store the *votes* that users have done. The order of the votes is not important, only the average and the information about who voted. If we would use a set that needs to be locked upon every modification, we would allow only one user to vote at the time, which is quite restrictive for a system.

The operations provided by value-sets are:

- *add(k, v)* - adds value v to the set associated to key k. If v is already in the set, the set remains unmodified.

- *remove(k, v)* - removes value v from set k. If the value was not in the set, the set remains unmodified.

- *readSet(k)* - returns all the elements added to the set, and that are not yet removed from it.

By using the Paxos-consensus protocol, we are able to propagate every *add/remove* operation to at least the majority of the replicas. The following table shows the evolution of adding elements *foo* and *bar* to the set associated to key k, where peers a, b and c are the responsible for the replicas of the set. The table shows which replicas were contacted each time under column TPs. We observe that peer a was contacted on both operations, but peers b and c where contacted only on one of them. At times t_2 and t_3, two different majorities are contacted, and in both cases it is possible to obtain the whole set by making a union of all sets.

Time	Operation	TPs	a	b	c
t_0	add(k, foo)	{a, b}	{foo}	{foo}	ϕ
t_1	add(k, bar)	{a, c}	{foo, bar}	{foo}	{bar}
t_2	readSet(k)	{a, c}	→ {foo, bar}		
t_3	readSet(k)	{b, c}	→ {foo, bar}		

Using the union of sets works only for adding operations. In the next example we introduce removals. An important rule on the implementation of remove, is that it can only be accepted by replicas who also have the value to be removed in their set. Otherwise, inconsistencies are introduced. In our example, the value *foo* has been accepted by peers a and b. Adding *bar* was successfully performed in all peers. However, removing *bar* at t_2 was only done at peers a and b. Having this knowledge, we can observe that it is impossible to know the correct state of the set only by looking at the elements stored on the majority of the peers, or even by looking at all of them, as in t_3. This is because peer c missed the removal of value *bar*. One way of eventually solving this inconsistency is by propagating every committed operation to all replicas. In such case, peer c would eventually receive operation *remove(k, bar)*. Nevertheless, this only provides us with eventual consistency when reading the set from the majority.

Time	Operation	TPs	a	b	c
t_0	add(k, foo)	{a, b}	{foo}	{foo}	ϕ
t_1	add(k, bar)	{a, b, c}	{foo, bar}	{foo, bar}	{bar}
t_2	remove(k, bar)	{a, b}	{foo}	{foo}	{bar}
t_3	readSet(k)	{a, c}	$\rightarrow$ {foo, bar}? or {foo}?		
t_4	readSet(k)	{b, c}	$\rightarrow$ {foo, bar}? or {bar}?		
t_5	readSet(k)	{a, b, c}	$\rightarrow$ {foo, bar}? or {foo}?		

Our solution is to store the operations instead of the values. On each transaction, the TM assigns an identifier to each operation, and every TP stores the operation together with its id. When *readSet(k)* contacts the majority, the set can be reconstructed from the union of all operations. We can observe in the next example how operations are stored on the transaction participants (only the id is shown at each replica to save space). We can observe the main difference with the previous example at time t_2. By storing the operations, peers a and b have a lot more information about the set, and when their information is combined with the operations of peer c at time t_3, the TM can discard the value *bar* and return the set with only *foo* on it, which is consistent with the global operations performed on the system.

Time	Operation	TPs	a	b	c
t_0	i:add(k, foo)	{a, b}	(i)	(i)	ϕ
t_1	j:add(k, bar)	{a, b, c}	(i, j)	(i, j)	(j)
t_2	j':remove(k, bar)	{a, b}	(i, j, j')	(i, j, j')	(j)
t_3	readSet(k)	{b, c}	(i, j, j') $\rightarrow$ {foo}		

It is very important to identify every operation, and to accept removals only after additions. If the following three operations are performed from different peers: *add(k, foo)*, *remove(k, foo)* and again *add(k, foo)*, depending on the order of arrival the value *foo* will be in the set or not. From the point of view of the decentralized system, no order can be guaranteed, but, it is important that all replicas reach a consensus about which is the order to follow by all of them. Therefore, it is necessary to identify one *add* from the other. Hence, the TM needs to provide an identifier for its operation, and conflict resolution must be achieved to define the order. Here, when we are working with the same value, there is a partial locking that we will discuss more in detail in the next sections.

5.5.1 The Transactional Protocol

Figure 5.5 describes the lock-free protocol for key/value-sets. Similarly to Paxos-consensus algorithm, the client sends one or more operations within a transaction to a TM. The TM creates an id for each operation, and registers the rTMs to start collecting votes from the TPs. Each TP is asked to vote on the operation it is involved. The only reason to reject an addition is that a concurrent transaction is trying to add the same value. Because operations are identified, it is necessary to only accept one. The only reason to reject a removal, it is that the value is not stored yet. There is a partial order between addition and removals for each value. This is to guarantee the reconstruction at reading time. Once TPs get the decision from the TM, they acknowledge each other the new operation. This is to guarantee that if a TP misses the transaction, it will eventually receive the operation from the other TPs.

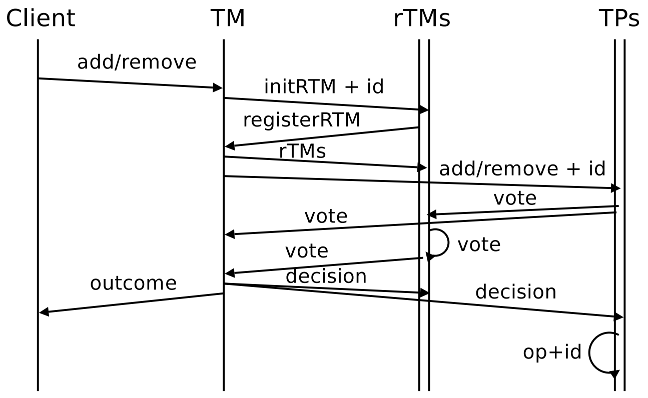


Figure 5.5: Transactional key/value-sets protocol

When two or more transactions try to concurrently add the same value to the same set, at least one of the TM will not collect a majority of successful votes. Instead of aborting the transaction as in Paxos, the TM will wait a random time to generate a new id and retry the operation. If the value has been finally added by another transaction, the TM will receive from the TPs votes suggesting that the value is *duplicated*. That means that the addition can be discarded, and the client is notified that its operation was successful, because the value is in the set now.

5.5.2 Semantics

Paxos and Two-phase commit have a very simple voting process depending on whether the lock is granted or not. TPs vote *commit* or *abort*, and the TM can decide upon the majority or totality, depending on the protocol. Value-Set protocol is simpler in the sense that it does not require a reading phase to establish a version of the item, because there is no version. But its voting mechanism is much more complex because it involves different possibilities.

Figure 5.7 shows the state diagram of a single value in a set, which helps to decide if an operation must be commited or not. In the case of abort, it does not have the same meaning as in Paxos. An abort in Value-Sets means that

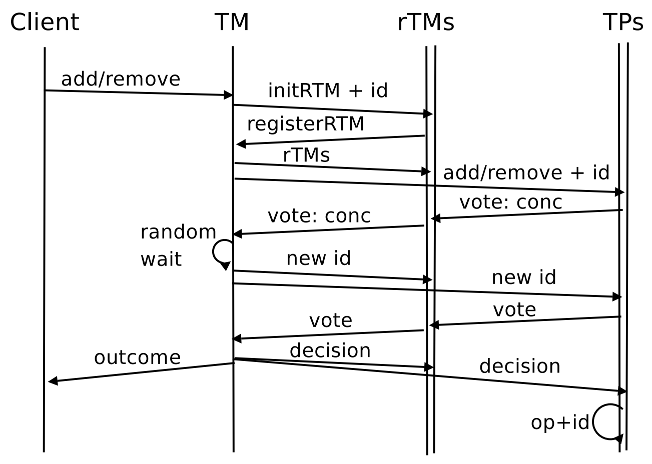


Figure 5.6: Retry when two clients attempt to concurrently add/remove the same value on the same set.

the operation is not stored on the TP, but the outcome for the client is always successful at the end.

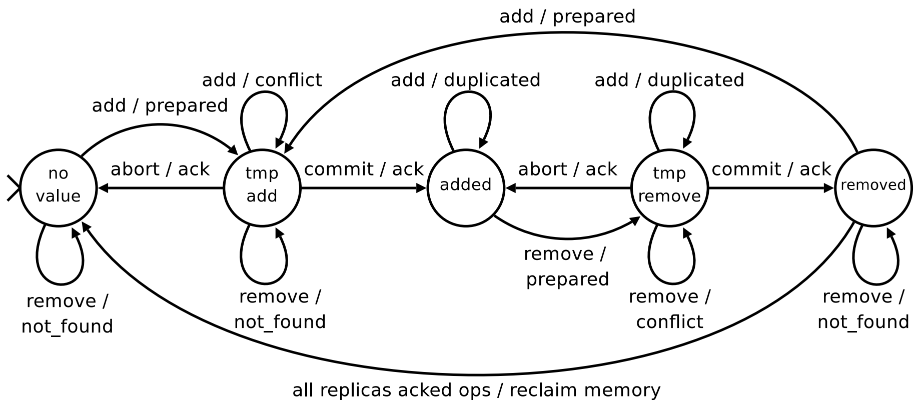


Figure 5.7: State diagram of a variable in a set.

Every value starts by not being in any set. Try to remove a value that is not in the set will result in a vote *not-found*. For the client, the outcome is successful because its removal ended up not having the value on the set. The only way of getting out of this initial state is by adding a value. The state *"tmp add"* lasts until the TM takes a decision. If the majority of TPs agrees on adding that particular operation, the transition to stated *"added"* is triggered by *commit*. If not enough TPs voted in favor of the addition, the transaction is *aborted* and retried. If another *add* operations arrives while the decision is being taken, it is informed of the conflict, which will make it try again later. If the second *add* arrives once the first one is already added, it will get the vote *duplicated*, meaning that the outcome for the client is affirmative considering that the value is in the set. Trying to remove a value that is not yet committed, results in *not-found*, basically because the values is not yet added to the set.

The only way to remove a value is with a transition from *"added"* to *"tmp remove"*. In such state, new additions are considered duplications because the item is technically still in the set, and concurrent removals are notified of the conflict to retry again later. Once the removal is committed, there is a state called *"removed"*, which is equivalent to *"no value"*. The main difference between both states is that *remove* provides important information to resolve conflicts with other replicas when reading the set. If another replica remains on the *added* state because of having missed the removal, staying in the state *remove* instead of *"no value"* will allow the TM to consistently rebuild the set. As a sort of garbage collector, when all TPs acknowledge the commit of the remove operation, all of them can reclaim the memory, and go back to the state *"no value"*.

5.6 Discussion on the Choice of Protocols

Two-phase commit is the most popular approach used by traditional databases to maintain replicas coherent in a distributed setting. On each transaction, two-phase commit strongly relies on the survival of the transaction leader, and it requires that all replicas become updated when the transaction completes. These two requirements are two hard to meet in dynamic systems such as peer-to-peer networks.

We have chosen the Paxos consensus algorithm because it relaxes these two conditions improving fault-tolerance with a low extra cost. First of all, Paxos does not rely on a single transaction leader because it uses a set of replicated transaction managers. If the majority of them survives, the transaction can terminate. Concerning the replicas, Paxos requires that only the majority of them agrees on the new value to commit the update. The quorum-based commit implies that read operations also need to read the data from the majority guaranteeing that the latest value is retrieved. Therefore, the requirements has been relaxed to provide fault tolerance without sacrificing strong consistency.

Paxos uses optimistic locking to provide concurrency control among transactions. This strategy works very well for asynchronous collaborative systems such as Wikipedia. However, it does not suit synchronous collaboration very well, because participants do not have any guarantee that their modifications will be committed to the storage. Therefore, we have adapted the protocol to support eager-locking. Like this, applications that need synchronous collaboration are also covered.

The basic data structure provided by DHTs it the key/value pair. It implies that data collections need to be implemented as set of key/value pairs, making it very costly to maintain. Another possibility is to implement them as a single value associated to a key. That implies that only one transaction can modify the collection, serializing concurrent operations. We introduce the key/value-set structure that relaxes two conditions: value-set provides an unordered data collection, and value-sets do not have versions. The relaxation allows us to design a lock-free protocol that allows concurrent modifications to a data collection. This protocol is entirely based on Paxos consensus algorithm, and it can be partially combined with Paxos and Eager Paxos.

These three protocols provide transactional support for a wide range of applications, and currently, they cover all the needs of the applications presented in Chapter 8. However, we cannot guarantee that they will be sufficient to cover all possible application scenarios. We specially identify the possible need of new protocols if eventual consistency is needed. Another data structure might also need the modification of the existing protocols.

The protocol that implements the notification layer is very simple and quite similar to those that implement publish/subscribe systems. We have decided to include it in Trappist to prevent busy waiting procedures at the level of applications. Without the notification layer, an application would have to periodically check the state of an item to know if its value has changed. This kind of behaviour is only needed for applications that automatically update the information about the state of the data. It is not needed for applications where the user decides which information is retrieved.

5.7 Replica Management

During the description of the transactional protocols, we have assumed that transaction participants are members of the replica set of each item, and that they are chosen by an underlying layer corresponding to the replica management. This layer needs to keep a consistent set of replicas even under churn. Systems such as Scalaris [SSR08] and DKS [Gho06] consider that the replica layer is completely orthogonal to the transactional layer. We understand it differently, because we see that restoring a lost replica needs the transactional support to know how to retrieve the latest value from the surviving replicas.

Let us analyse more deeply replica management. The replica set of each item is formed with f replicas. One of the problems we can encounter is that $f+1$ peers claim to hold a valid replica. There are a couple of things to consider here. First of all, why is it a problem to have $f + 1$ replicas? The problem is that you could potentially have two majorities with respect to f. In Beernet, as in Scalaris, we take f as an even number. Like that, you will need $f + 2$ replicas to have two majorities. With $f + 1$ is not enough to break majority if f is an even number.

Second question is, how can the system end up having $f + 1$ replicas? The first possible cause is lookup inconsistency: two nodes thinking that they have to store a replica. Reducing the amount of lookup inconsistencies is a way of addressing this issue, and this is what we do with the relaxed ring. The second way of having $f + 1$ replicas is churn. The only problem actually comes when there is churn affecting the responsibility of one of the transaction participants. In that case, following the more detailed description of Paxos consensus algorithm [MH07], the items involved in the transaction are actually locked, and they are not transfered to the new responsible until the transaction has finished.

One suggestion to avoid two majorities of replicas is to add group management to the replica sets. This idea can become too expensive to implement and maintain because there is a replica set per each stored item. It is true that because of symmetric replication many replica sets overlap, but there is no guarantee on this property to define a cost-efficient group maintenance. We consider that there is no need for group management on symmetric replication. Peers do not store references to every peer in the replica set of every item or replicated item is stored in the peer. It would be too expensive. This is actually one of the advantages of symmetric replication: when a peer needs to find the other replicas, it uses the symmetric function and lookup requests to identify the other members of the set.

Let us consider that there is group management for the replica set to make the replica layer completely orthogonal to the transactional layer. When there is churn, the peer that takes over a range of responsibility of one of the members of the replica set, needs to read from the majority of the replicas to decide the value of the items it is going to host. Therefore, some transaction is still needed anyway when a new peer *"joins the replica set"*. We discuss now two possible ways of getting a new peer in the replica set: a new peer joins the network, and a peer fails and it is replaced by the recovery mechanism.

5.7.1 New Peer Joins the Network

Symmetric replication was introduced in [Gho06], without discussing any transactional support. To maintain the replica set, the new joining peer should ask its successor for the values of the items its storing. Note that the new peer replaces its successor as member of the replica set of a certain amount of items. Asking the successor is fine, but it assumes that all replicas are up to date. If the successor does not have the latest value, it does not introduce real problems because it would replace a bad replica with another bad replica. Not a good replica for a bad one. Still, it might be a good idea to retrieve the value of the item from the replica set to replace a bad replica by a good one. However, performing read from majority every time a new peer joins the network is expensive.

5.7.2 Failure Handling

Whenever there is failure, DKS [Gho06] makes use of the *startbulkown* operator which is in charge of finding the owner of a replicated key, and retrieve the values from that peer. We believe that here we arrive to a similar analysis of the joining peer: is it enough to retrieve the items from only one node? In the case of the failure recovery it is more important to read from the majority, because the recovery node cannot know if the dead peer was holding the latest value or not. Let us say the replica set is form by peers a, b, c, d and e, where a, b, and e have the last up to date value of item i. Therefore, c and d hold and old value of i. Let us suppose now that peer e dies, and f takes over its responsibility. Peer f reads i from c or d, and then, the system ends up having a majority c, d and f holding an old value of i, which is incorrect. This problem can already be improved by choosing an even amount of replicas, but if the read is done from the majority, it does not matter if the replication factor is odd or even.

5.8 Trappist

As we have previously mentioned in this chapter, the transactional layer implementing these three protocols is called Trappist, which stands for Transactions over peer-to-peer with isolation, where isolation means that transactions are atomic and with concurrency control. In this section we show how to use the transactional support of Beernet, which is implemented with the Mozart [Moz08] programming system. By describing Trappist's API, we also analyse the high level abstractions provided by the system, and how replica maintenance is hidden from the programmer. We start by creating a Beernet peer, which can be built with with or without the transactional support. To make it explicit, the following example set the 'transaction' option to **true**:

```
functor
import
   Pbeer at 'Pbeer.ozf'
define
   Node = {Pbeer.new args(transactions:true)}
```

The most basic support provided by Beernet corresponds to the DHT operations *put* and *get*. These operations do not replicate the value of the item, but they are also part of the implementation of the transactional layer which actually realizes the replication. What follows is an example of how put and get can be used.

```
{Node put(key value)}
Value = {Node get(key $)}
```

To use the transactional layer, the user must write a procedure with one argument, typically named *Obj*. This argument represents a transactional object, which is an instance of the transaction manager that triggers the transaction. The object receives the operations *read* and *write*, which are almost equivalent to *put* and *get*. The main semantic difference between the operations is that if the transaction is aborted, *write* has no effect on the stored data. And if the transaction succeeds, the value is written at least on the majority of the replicas. Other operations received by the transactional object are *commit* and *abort*, to explicitly trigger those actions on the protocol. The operation *remove* is also implemented to delete an item from the DHT.

To run the transaction, the user must invoke the method *runTransaction*, which receives three arguments: the procedure containing the operations, a port to receive the outcome of the transaction, and the protocol to be used for running the transaction. Note that at the creation of the node, we did not specify the protocol to be used by every transaction. This is because the protocol can be chosen dynamically, allowing the users to choose the best suitable protocol for every functionality. Algorithm 15 is a complete example for writing two items with key/value pairs: *hello*/*"Charlotte"* and *foo*/*bar*. The outcome of the transaction appears on variable *Stream*, which is the output of port *Client*. If the outcome of the transaction is `commit`, it guarantees that both items where successfully stored at least in the majority of the correspondent replicas.

Algorithm 15 Using transactions with Paxos consensus to write two items

```
declare
Stream Client
Trans = proc {$ Obj}
       {Obj write(hello "Charlotte")}
       {Obj write(foo bar)}
       {Obj commit}
    end
{NewPort Stream Client}
{Node runTransaction(Trans Client paxos)}
if Stream.1 == commit then
  {Browse "transaction succeeded"}
end
```

To retrieve the values the user passes a variable which has no value yet. The value is bound by the transactional object. Algorithm 16 shows how to retrieve the values stored under keys *hello* and *foo*.

Note that it is not necessary to catch exceptions using Beernet, because the outcome is reported on the stream of the client's port. If there is a failure on the transaction, the outcome will be `abort`, and the user will be able to take the corresponding failure recovery action. If the item is not found, the variable used to retrieve the value is bound to a *failed value*. This language abstraction will raise an exception whenever is used. Like this, exceptions are

Algorithm 16 Using transactions with Paxos consensus to read two items

```
declare
V1 V2
Trans2 = proc {$ Obj}
       {Obj read(hello V1)}
       {Obj read(foo V2)}
     end
{Node runTransaction(Trans2 Client paxos)}
{Browse "for hello I got"#V1}
{Browse "for foo I got"#V2}
```

triggered in the calling site, and not at any of the peers. Now, to prevent catching exceptions when using the value, the Mozart programming system provides Boolean checkers to test whether a variable is bound to a failed value or not.

5.9 Conclusion

In this chapter we analysed the strong requirements of Two-phase commit, which are too hard to meet in peer-to-peer systems due to its dynamism. We have chosen Paxos consensus algorithm as the base protocol for Trappist, because it has relaxed conditions about storage and fault tolerance. It relaxes the storage because it only needs the agreement of the majority of the replicas instead of requiring all of them to agree. It relaxes fault tolerance because it does not rely on a single transaction manager because it uses replicated managers.

To support a wider range of applications, we have extended Paxos with a modified protocol that provides eager-locking of items. This protocol suit better synchronous collaborative applications, whereas Paxos works better for asynchronous systems. Both protocols can be used in combination with a notification layer that informs the readers of an item whenever the item is locked or updated.

Following the relaxed approach, we have relaxed data collections to provide a lock-free protocol for key/value-sets. This data abstraction stores an unordered set of values associated to a single key, without having a versioning system. The lack of order and versioning is the relaxation that allows us to get rid of locks on sets.

Trappist works independent of the replication strategy and the SON that supports the communication between peers. We will review Beernet's architecture more in detail in Chapter 7, where we will see how is possible to achieve this kind of modularity. More discussion about the use of Trappist's protocols will be presented in Chapter 8, where we will described the applications built on top of Beernet. Evaluation of the protocols is presented in Chapter 6.

Chapter 6

Evaluation

'Stop. What... is your name?'
'It is Arthur, King of the Britons.'
'What... is your quest?'
'To seek the Holy Grail.'
'What... is the influence of non-transitive connectivity in the peer-to-peer ring?'
'What do you mean? A perfect or relaxed ring?'

freely adapted from
"Monty Python and the Holy Grail"

After the analysis we have done about the algorithms and self-management behaviour of the relaxed ring, we do now the empirical evaluation comparing it with other networks, specially with Chord [SMK$^+$01]. We start by describing CiNiSMO, our concurrent simulator, and then describe the results we have obtained measuring cost-efficient ring maintenance, lookup consistency, size and amount of branches, and efficient routing. We also analyse the performance of different transactional protocols putting enphasis on the lock-free key/value-set protocol. A deep analysis on the impact of network address translation (NAT) devices is also included in this chapter.

6.1 Concurrent Simulator

CiNiSMO is a Concurrent Network Simulator implemented in Mozart-Oz. It has been used for evaluating the claims made about the relaxed ring in Chapter 3, and we continue to use it for ongoing research with other network topologies. In CiNiSMO, every node runs autonomously in its own lightweight thread. Nodes communicate with each other by message passing using ports, and cannot inspect the state of other nodes as if it were a local operation. We

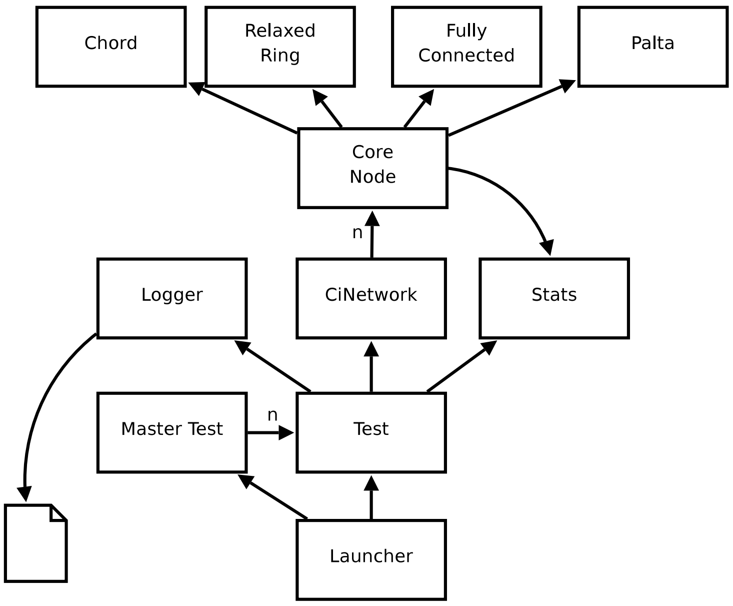


Figure 6.1: Architecture of CiNiSMO.

consider that these properties give us a good simulation of real networks. We have released it as a programming framework that can be used to run other tests with other kinds of structured overlay networks. Another motivation for releasing CiNiSMO is to allow other researchers to reproduce the experiments we have run to generate our conclusions.

The general architecture of CiNiSMO is described in Figure 6.1. At the center, we observe the component called "CiNetwork". This one is in charge of creating n peers using the component "Core Node". The core node delegates every message it receives to another component which implements the algorithms of a particular network. Currently, we have implemented in CiNiSMO the relaxed ring [MV08], Chord [SMK+01], Fully connected networks and Palta [CMV+08]. To add a new kind of network to this simulator it is sufficient to create the correspondent component that handles the messages delegated by the core node.

Every core node transmit information about the messages it receives to a component called "Stats", which can summarize information such as how many lookup messages were generated, or how many crash events were triggered. The component that typically demands this kind of information is the "Test". This is another component that can be implemented to define the size of the network and the kind of event we want to study. Only one CiNetwork is created per Test. When the relevant information is gathered, it is sent to a "Logger", which outputs the results into a file.

Since it is cumbersome to run every test individually many times, it is possible to implement the component called "Master Test", which can organize the execution of many testing, changing the seed for random generation numbers, or a parameter that is used for the creation of the CiNetwork. The software can be dowloaded at `http://beernet.info.ucl.ac.be/cinismo`, with documentation on how to use it.

6.2 Branches in the Relaxed Ring

To test the amount of branches that appear on a network, we have bootstrapped networks with different sizes, and using different seeds for random number generation. Every networks starts with a single node, and peers are constantly joining until the network reaches the desire size. Even though there are no failures, the joining activity generates a considerable amount of churn. Since the relaxed ring maintenance is based on correction-on-change, there is no rate we can provide for the churn until we compare it with Chord rings, because then we introduce periodic actions that can be compared with churn.

Figure 6.2 shows the amount of branches that can appear on networks with 1,000 to 10,000 nodes. The coefficient q represents the quality of network's connectivity, where $q = 0.95$ means that when a node contacts another one, there is only a 95% of probability that they will establish connection. A value of $q = 1.0$ means 100% of connectivity. On that value, no branches are created, meaning that the relaxed ring behaves as a perfect ring on fault-free scenarios. The worse case corresponds to $q = 0.9$. In that case, we can observe that the amount of branches is less than 10% of the size of the network, as expected. Consider peers i and k, where i is the current predecessor of k. If they cannot talk to each other, k will form a branch. If another peer j joins in between i and k having good connection with both peers, the branch disappears.

On the contrary, if a node l joins the network between k and its successor, it will increase the size of the branch, decreasing the routing performance. For that reason, it is important to measure the average size of branches. If message *hint*, explained in section 3.8, works well for peer l, then, the branch will remain on size 1. Having this in mind, let us analyse figure 6.3. The average size of branches appears to be independent of the size of the network. The value is very similar for both cases where the quality of the connectivity is poor. In none of the cases the average is higher than 2 peers, which is a very reasonable value. If we want to analyse how the size of branches degrades routing performance of the whole network, we have to look at the average considering all nodes that belong to the core ring as providing branches of size 0. This value is represented by the curves *totalavg* on the figure. In both cases the value is smaller that 0.25. Experiments with 100% of connectivity are not shown because there are no branches, so the average size is always 0.

The chosen values for quality q are a bit worse than some real measurements on the internet. The King Data Set [GSG02] has 0.8% of the nodes that cannot connect with each other. According to [FLRS05], 2.3% of all pairs of nodes

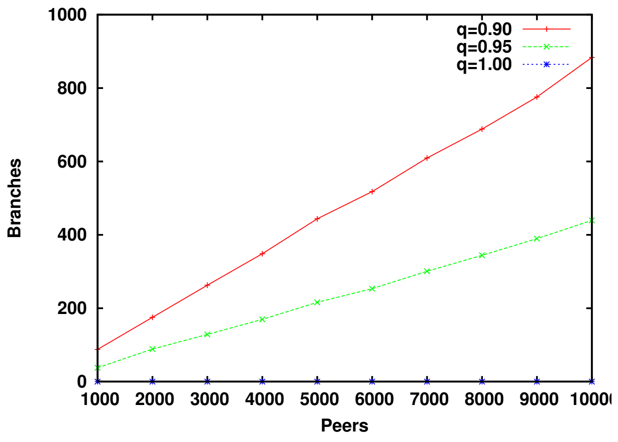


Figure 6.2: Average amount of branches generated on networks with connectivity problems. Networks where tested with peers having a connectivity quality q, representing the probability of establishing a connection between peers, where $q \in \{0.9, 0.95, 1\}$.

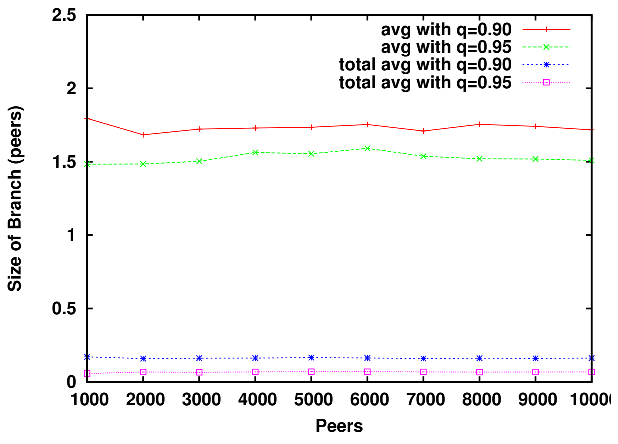


Figure 6.3: Average size of branches depending on the quality of connections: *avg* corresponds to existing branches and *totalavg* represents how the whole network is affected.

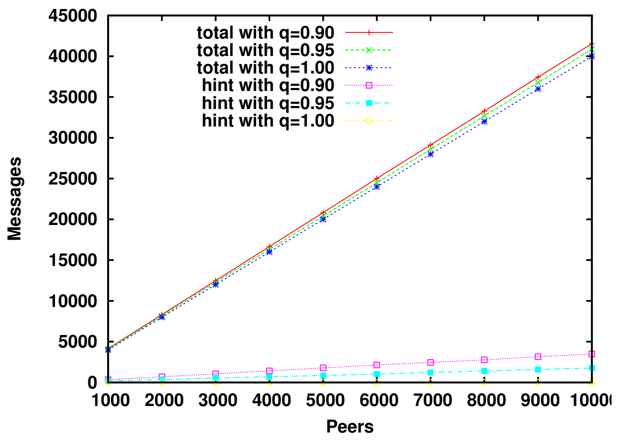


Figure 6.4: Number of messages generated by the relaxed ring maintenance. Three curves labeled *total* represent the total amount of messages exchanged between all peers depending on the connectivity coefficient. Curves labeled *hint* represent the contribution of *hint* messages to the total amount.

in PlanetLab [The03] cannot talk to each other. Therefore, our values for coefficient q seems to be reasonable to study the relaxed ring.

6.3 Bandwidth Consumption

In this section we try to answer the following questions: How many messages are exchanged by peers to maintain the relaxed ring structure? How much is the contribution of the *hint* messages to the bandwidth consumption so that branches are kept short? These questions are answered in figure 6.4. We can observe that the amount of messages increases linearly with the size of the network keeping reasonable rates. The fault-free scenario has no *hint* messages as expected, but the total amount of messages is still pretty similar to the cases where connectivity is poor. This is because there are less normal join messages in case of failures, but this amount is compensated by the contribution of *hint* messages. We observe anyway that the contribution of *hint* messages remains low.

6.4 Comparison with Chord

As we previously mentioned, we have also implemented Chord in our simulator CiNiSMO. Experiments were only run in fault-free scenarios with full connectivity between peers, and thus, in better conditions than our experiments with the relaxed ring. The idea is to respect the assumptions made by

Chord authors as much as possible. We make our comparison considering two parameters: lookup consistency and bandwidth consumption.

6.4.1 Lookup Consistency

Even though the connectivity conditions of the experiments running Chord were much better than those of the relaxed ring, we observed many lookup inconsistencies on high churn. To reduce inconsistency, we trigger periodic stabilization on all nodes at different rates. The best results appeared when only 4 nodes joined the ring in between every periodic stabilization. The amount of nodes joining the ring during that period is what we call stabilization rate. As seen in figure 6.5, the larger the network, the less inconsistencies are found. An inconsistency is detected when two *reachable* nodes are signalled as responsible for the same key. We can observe that stabilization rates of 5 converges pretty fast to 0 inconsistencies. Stabilization every 6 new joining peers only converge on networks of 4000 nodes. On the contrary, rate values of 7 and 8 presents immediately a high and non-decreasing amount of inconsistencies. Those networks would only converge if churn is reduced to 0. These values are compared with the worse case of the relaxed ring (connectivity factor 0.9) where no inconsistencies where found. In this experiment, every instance of a Chord network of a given size was run with six different random number generators. What it is shown in the graph is the average of those instances.

6.4.2 Bandwidth Consumption

We have observed that lookup consistency can be maintained in Chord at very good levels if periodic stabilization is triggered often enough. The problem is that periodic stabilization demands a lot of resources. Figure 6.6 depicts the load related to every different stabilization rate. As expected, the highest cost corresponds to the most frequently triggered stabilization. Observing Figures 6.5 and 6.6, it seems that the cost of periodic stabilization pays back in networks with sizes until around 3000 nodes, because the level of lookup consistency is still good. But, this cost seems too expensive with larger networks.

In any case, the comparison with the relaxed ring is considerable. While the relaxed ring does not pass 5×10^4 messages for a network of 10000 nodes, a stabilization rate of 7 on a Chord network, starts already at 2×10^5 with the smallest network of 1000 nodes. Figure 6.6 clearly depicts the difference on the amount of messages sent. The point is that there are too many stabilization messages triggered without modifying the network. On the contrary, every join on the relaxed ring generate more messages, but they are only triggered when they are needed.

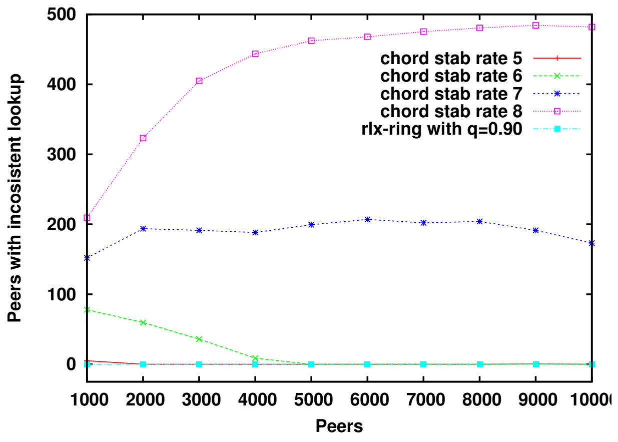


Figure 6.5: Amount of peers with overlapping ranges of responsibilities, introducing lookup inconsistencies, on Chord networks under different stabilization rates for different network sizes. Comparison with the relaxed ring with a bad connectivity. The stabilization rate represents the amount of peers joining/leaving the network between every stabilization round. The value of zero in the Y-axis has been raised to spot the curve of the relaxed ring and Chord with a very frequent stabilization rate equal to 5.

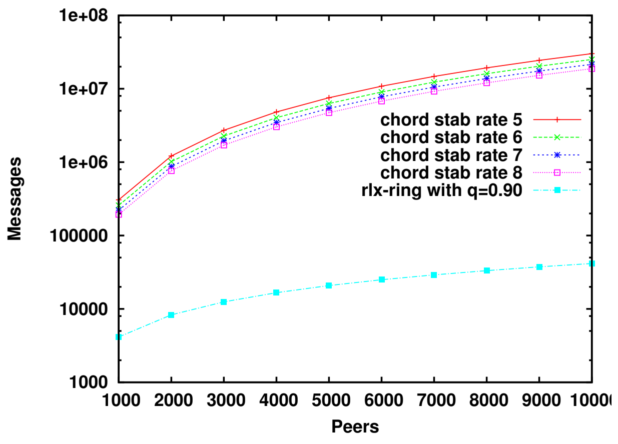


Figure 6.6: Load of messages in Chord due to periodic stabilization, compared to the load of the relaxed ring maintenance with bad connectivity. Y-axis presented in logarithmic scale.

6.5 Efficiency of the Routing Table

This section presents an analysis of the results obtained by simulating PALTA, the self-adaptable finger table we described in Section 3.9. This finger table uses a full mesh graph when the estimated size of the network is smaller than a given value ω. When this threshold is reached, the finger table of new peers will only work with DKS fingers. If network's size decrements back below the value of ω, all peers with adapt their finger tables to get closer to a full mesh, taking more advantage of the connectivity.

To validate this finger table strategy, we simulate with two different values of ω, and we compare them with the relaxed ring using plain DKS, and with fully connected network, which always have full mesh connectivity. To measure the efficient use of resources, we have measured the average amount of active connection a node has in every of these networks. To study the performance of the topologies, we have measured the total amount of messages needed to build the network, and the average hops needed to perform a lookup.

Every topology is tested by building networks from 20 to 1000 nodes, increasing the size by 20 nodes at every iteration. Plotted values represent the average of running every experiment with several seeds for random number generation. In the case of PALTA, we tested the algorithm using two different values for ω, being 100 and 200. Reaching 1000 nodes might be considered not large enough for large scale networks, but it is enough to observe the behavior after the ω threshold is reached and extrapolate the scalability from the curves obtained.

6.5.1 Active Connections

One of the goals of PALTA is to dynamically adapt its topology to optimize the use of the network. For small networks, that means that we want to directly connect as much peers as possible to reach every peer in the minimum amount of hops. Small is defined in terms of the ω value.

Figure 6.7 shows the average amount of active connections per peer in the different topologies. We can observe that the *fully connected* network increments the amount of connections linearly, and therefore, it does not scale at all. Part of the curve is missing, but it clearly corresponds to $n-1$, being n the size of the network, because every node is connected to all the other peers. As expected, the *relaxed ring* with plain DKS appears as the topology where peers manage the smallest amount of connections, showing that it has good scalability for large networks. Let us analyze now the behavior of PALTA. In both cases, with ω 100 and 200, we observe that the amount of connections increases linearly as a fully connected network until reaching ω peers. From that point on, the average of connections decreases very fast, converging asymptotically to the values of the relaxed ring. This is because all new nodes that join the network after the threshold of ω is reached, create only the amount of fingers needed by a relaxed ring. In fact, ω peers manage $\omega-1$ connections, and $N-\omega$

peers manage k fingers, with N being the size of the network. Meaning that the larger the network, the smallest the average. Of course, this decreasing behavior continuous until it almost reaches the curve of the relaxed ring, then, the average can only increase according to the size of the network.

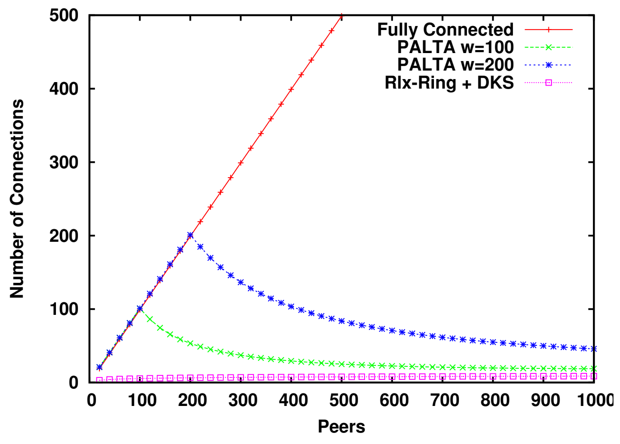


Figure 6.7: Average amount of active connections vs number of peers. This chart evaluates if a small network take advantage the proximity by opening more networks, and it validates the scalability of the solution.

In conclusion, Figure 6.7 shows us that PALTA uses actively more resources than a regular ring, but it is capable of self-adapting when the network becomes too large and provide a good scalability.

6.5.2 Network Traffic

When peers enter in a distributed network, they generate a number of messages to correctly join without leading the network to an unstable state. In the case of a *fully connected* network, the joining peer will always need $2 * n$ messages to contact all peers in a network of size n. Therefore, the cost of a new joining peer increases as the size of the network increases. In our simulation we contact directly every peer. In case a broadcast mechanism is used to propagate the *join* of a new peer, n messages are needed to reach every peer, plus n message to acknowledge the new peer, making $2 * n$ messages. These measurements about fully connected network do not apply to small networks using Bluetooth, where *one hop* broadcasting is available. On the other hand, Bluetooth cannot currently handle the amount of connection we are testing. They actually belong to a different problem domain.

In the relaxed ring with plain DKS, the joining peer needs to send messages for contacting the predecessor, successor and the k fingers. Therefore, the marginal cost of a joining peer is almost independent of the size of the network.

The only difference occurs with the amount of messages needed for localizing the k finger, which increases logarithmically with respect to the size of the network, as we will see in Section 6.5.3.

Figure 6.8 does not show the marginal cost of joining a network, but the total amount of messages generated to construct every network we have studied in section 6.5.1. We can see that with less active connections, as in PALTA or the relaxed ring, the number of messages remains small, generating less network traffic. The curve of the fully connected network increases quadratically, generating $n * (n - 1)$ messages, with n being the size of the network, we can conclude that this network cannot scale.

The curve of the relaxed ring with DKS shows a constant and controlled increment in the amount of messages, keeping them at a very low rate, showing that it scales very well. Now, the results obtained from experiments with PALTA are very interesting because both perform better than the ring for larger networks. One can observe that PALTA with $\omega = 100$ and $\omega = 200$ increases quadratically the amount of messages, as in a fully connected network. This happens only until the network reaches a size of ω peers. Then, the amount of messages increases slower that in a ring, and furthermore, after a certain size of the network, both PALTA networks remain at better values that the relaxed ring. The explanation for this is that when a new peer join in the network, it needs less messages to find the k fingers. This is because PALTA has ω peers with a larger routing table ($\omega > k$), making a more efficient jump during the routing process. We study this further in the following section.

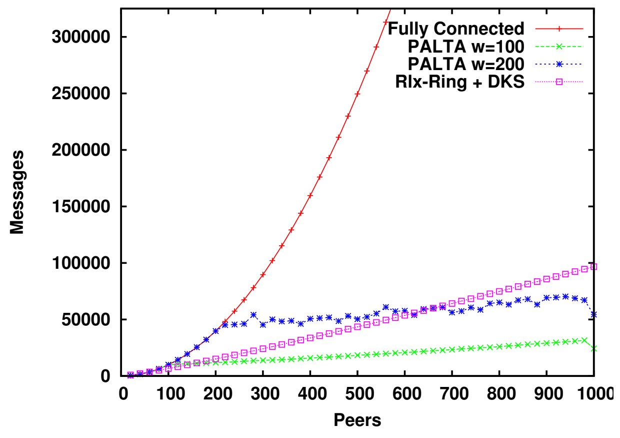


Figure 6.8: Total amount of messages to build the network *vs* number of peers. This chart evaluates how costly is every network. It is interesting to see that PALTA networks start by behaving as a fully connected network, but then they are more efficient than the relaxed ring with DKS. This is explained in the next chart.

This means that the cost of maintaining a small fully connected network can help a larger network to be more efficient for routing, generating less network traffic. We observe that PALTA could not only be used for ambient intelligent networks as it was in their original conception, but also as the topology for large scale systems. This is why we have actually adopted PALTA for the implementation of Beernet.

6.5.3 Hops

To confirm our conclusions from the previous experiment, we decided to measure the average amount of hops needed for a message to reach its destination. This is known as a *lookup operation* in a ring. This experiment does not consider fully connected networks, because there is no concept of responsibility is such systems. In addition, because of its characteristics, peers in a fully connected network reach any other peer in the network in 1 hop.

In Figure 6.9 we can observe the results obtained. The relaxed ring with DKS fingers shows that the number of needed hops increase logarithmically when the network size increases. PALTA performs better than the relaxed ring due to fact that some peers have a larger routing table, confirming the results from the previous experiment. In both cases, PALTA presents an average number of hops slightly smaller than 2 if the network consist of less than ω peers. This is because the network is fully connected, and therefore, in can reach the predecessor of the responsible of the looked up key in only one hop. The second hop is needed to reach the responsible. The average is smaller than 2 because the randomized experiments sometimes generates lookups where the responsible is the peer triggering the lookup.

After the value of ω is reached, the average increases faster in PALTA with $\omega = 100$ than with $\omega = 200$. This is clearly due to the amount of peers having a larger routing table. We observe that in both cases the system behaves much better than the ring. We expect that for larger networks the value would converge to the curve of the ring, but still performing better. What we cannot currently explain is the behavior of PALTA with $\omega = 100$ when the network is in between 100 and 200 nodes. It seems to perform even better than a $\omega = 200$.

Something that we still need to investigate is the construction of a network where every peer defines its own ω value according to its own resources. That behaviour would allow us to use PALTA in networks formed by heterogeneous devices, where each one of them is limited by its own resources.

6.6 Trappist's Performance

It has been claimed in Chapter 5 that Paxos-consensus algorithm can provided transactional support to DHTs without strong consistency and with a reasonable cost in performance. It is also claimed that key/value-sets can beat the performance of key/value pairs thanks to the relaxation in ordering and

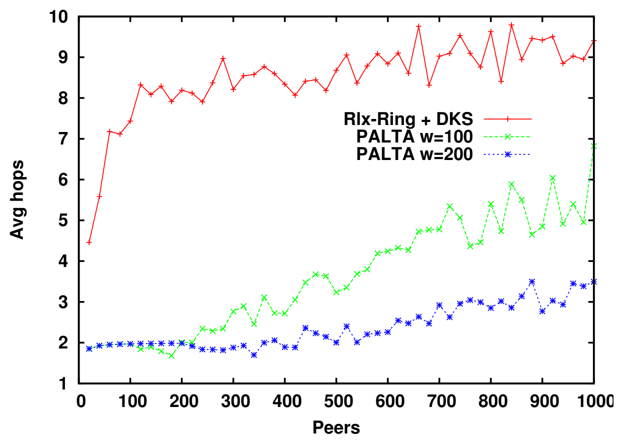


Figure 6.9: Average number of hops vs number of peers. The relaxed ring with DKS fingers follows a logarithmic distribution in the amount of hops that are needed to reach any peer in the network. PALTA network converge to this value for large networks, but they remain lower because there are some peers with more connections than the average peer in the network, providing a more significant jump to complete the lookup request.

versioning, deriving in a transactional protocol without distributed locks. To verify those claims, we have implemented two-phase commit, Paxos-consensus, and transactional key/value-sets directly in Beernet. Details on the implementation architecture will be described in detail in Chapter 7. In this section we will study the performance and scalability of the system, and we will compare the performance of concurrent modifications to a single set.

6.6.1 Scalability

To measure scalability, we have built a Beernet network using one of the student labs of our Computer Science Department. Details about the hardware used will be explained when we compare the measurements to Scalaris [SSR08]. For this experiment, we use a network with only 16 nodes and replication factor $f = 4$. The size of 16 nodes was chosen according to the maximal amount of machines we had available at the lab, so that the best performance is achieved by having only one node per processor.

Figure 6.10 shows the average performance of three protocols implemented in Beernet: Paxos consensus algorithm, key/value-sets and Two-phase commit. The performance is given in transactions per second. Every transaction consisted in reading an item, incrementing its value, and writing the new item. Running the experiment in one machine means that all 16 nodes run in a single processor. When the experiment was run in two machines, the network was

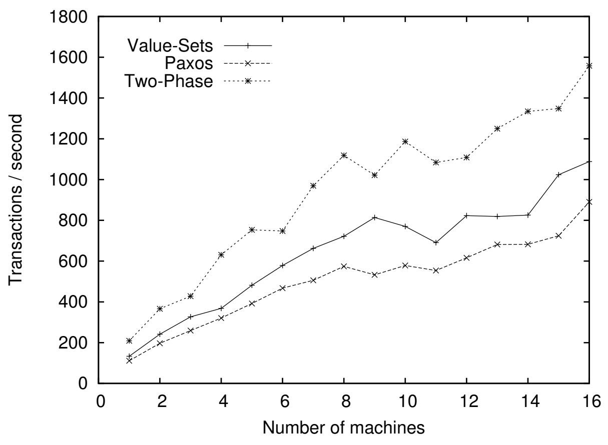


Figure 6.10: Average performance of Trappist's protocols. The amount of transactions performs per second increments with the amount of machines in use.

split so that 8 nodes run in each processor. The progression of machines was done building the network as balanced as possible, finally reaching 16 machines, each of them running a single peer in the network. Every test was run 10 times to obtained the average.

We can observe Figure 6.10 that Two-phase commit performs better than Paxos and value-sets. This result is expected because Two-phase commit does not work with replicated transaction managers, it has one round less of messages, and in total, it sends $2f^2$ less messages that Paxos. However, the test does not include failures, and therefore, the evaluation of Two-phase commit is the best possible, because the protocol does not tolerate failures very easily. The goal of measuring this protocol is to understand the cost of using replicated transaction managers, and to establish a maximum performance to target. According to Scalaris measurements [SRHS10], it is possible to tune Paxos so that it gets closer to Two-phase commit performance. We see this as part of our future work.

An interesting result that we observe on the plot is that performance improves immediately by adding a second machine. The influence of the network latency in the performance is much less than the gain in parallelism by using two processor to run the network. A nice consequence is that adding more nodes keeps on improving the performance of the system, highlighting its scalability.

The better performance of Value-sets over Paxos is consistent during the whole experiment. This result confirms our claim that using lock-free protocols improves the performance of the protocol.

Figure 6.11 adds the standard deviation to the average results presented

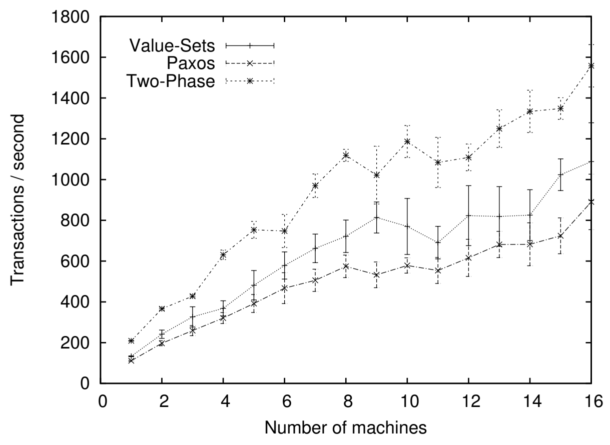


Figure 6.11: Average performance of Trappist protocols including standard deviation. The difference between the three protocols is consistent through the different amount of machines.

in Figure 6.10. We can observe that the data is consistent though the whole experiment showing that Two-phase commit is constantly better that Value-Sets, which is constantly better than Paxos.

We have mentioned several times that Scalaris is the closest related work to Beernet's Trappist layer. Their measurements in performance and scalability presented in [SRHS10] inspired our validation tests, and therefore, we would like to analyse how the results are related. First of all, it is not possible to really compare the numbers because of the difference in hardware and network speed, but we provide some information to understand the differences.

The numbers shown in [SRHS10] are the best performance results Scalaris could obtained. Therefore, we have built Figure 6.12 with the best performance we obtained. Scalaris used a cluster with 16 nodes. Each node has two Dual-Core Intel Xeon running at 2.66 GHz and 8 GB of RAM. That makes 4 cores in total per machine. In our student lab, each node has a single core Intel Pentium 4 running at 3.2 GHz and 1 GB of RAM. That makes 1 core per machine. A more important difference is that Scalaris run on a cluster where nodes where connected via GigE providing 1,000 Mbps. Our machines where connected via LAN with network cards running at 100 Mbps.

For the case of Paxos, the best number after 10 experiments running on 16 machines was 1,114 transactions per second. For Value-sets, we were able to run 1,350 transactions per second. We should not compare this number with 16 machines on Scalaris because each of them has 4 cores. Therefore, we should look at their results with 4 machines which is close to 5,000 transactions. Their improved version of Paxos reduces the 3 coordination rounds to 2, making it

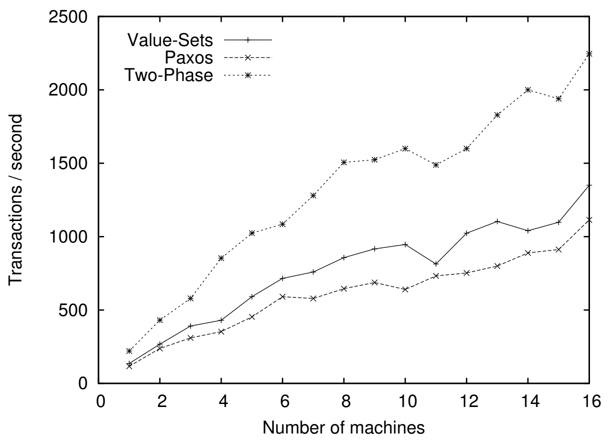


Figure 6.12: Best performance obtained. We can observe that Paxos passed the line of 1,000 transactions per second with 16 machines. Key/Value-Sets did it already with 12 machines.

already a factor of 1.5 better. We can guess that the remaining factor of difference might be influenced by the difference in communication speed. Their nodes not only communicate faster in the cluster, but they have groups of 4 cores running on the same machine. In fact, their performance measurements shows that 2 nodes run slower that a single node with 4 cores. That means that network latency is really an issue. In our results, we just get improvements in performance by adding nodes. Another point to consider is that Scalaris tests were run with exclusive access to the cluster, whereas the student lab is a shared resource in our department, were the machines are constantly used to provide grid computing.

Coming back to our comparison between Paxos and Two-phase commit, despite the disadvantage in speed, we know that Two-Phase commit is much less robust than Paxos, and therefore, the extra cost pays off. Furthermore, there is an important advantage of using the majority of the replicas to commit the changes, instead of all of them as we will see in the next section.

6.6.2 Concurrent Modifications

The plot presented in Figure 6.13 spots the main advantage of using consensus, relaxing order of values in a set, and as consequence, avoiding locks to store items. For this experiment we come back to our simulation environment due to the limitations of the size of the computer lab. For these measurements we use networks of 256 peers and we work with data collections implemented using key/value pairs and key/value-sets. The former ones are modified using Paxos

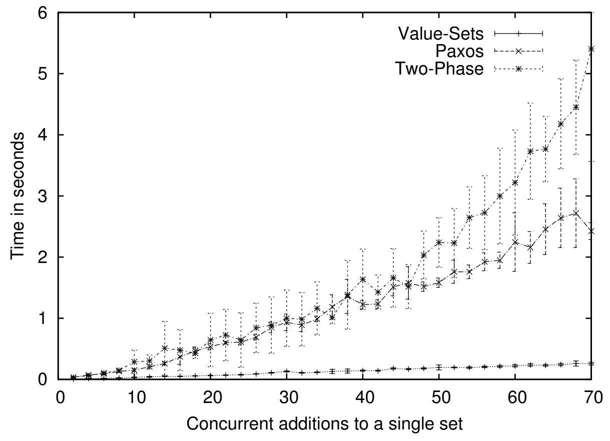


Figure 6.13: Concurrent transactions over the same set.

and Two-phase commit, whereas the latest are modified using out lock-free protocol for unordered set without versions. The test consists on performing N concurrent additions to a single set, measuring the needed time to complete all modifications. Each transaction was initiated on a different peer as with the previous measurements. Every time a transaction was aborted because of not obtaining the necessary locks, the transactions was retried after waiting a random time less than 42 milliseconds. In previous tests we eagerly retried every transaction with really bad results for Paxos and Two-phase commits. For instance, Two-phase commit was taking around 15 seconds to commit 40 transactions. With the new strategy of waiting a random time, 40 modifications to a set takes less than 2 seconds using Two-phase commit.

Figure 6.13 also includes the standard deviation of the average times. We can observe that Two-Phase commit consistently has the worse performance because it needs to gather all locks from all replicas. Therefore, its chances to succeed are less than Paxos. Starting from around 46 concurrent modifications, Paxos performs much better because it only needs to gather the majority of the locks to commit. The clear advantage can be observed in the curve of the Value-Sets. Its performance is not affected when running several transactions on the same item and it remains very stable across the whole experiment. Even with 70 modification to a single set, it average 258 milliseconds.

To see how close the simulation results can be to real networks, we performed the same test in the computer lab using the two extreme protocols: two-phase commit and value-sets. The main difference is that networks are composed by only 16 nodes running on 16 machines communicating through LAN protocol. Given that, we tested from 3 to 15 concurrent modifications to the same sets, with increments of 3 for each test. The average are depicted

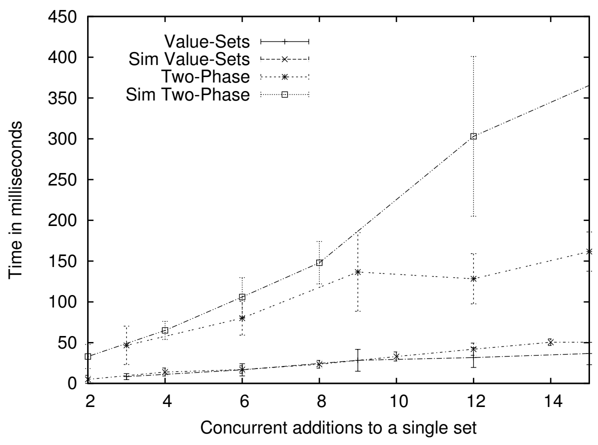


Figure 6.14: Comparing simulations with results obtained running an equivalent experiment on the computer lab.

in Figure 6.14, together with the simulation results. We can observe that results are consistent and closely related. Value-sets still performs better than Two-phase commit working even a bit faster than our simulation results. Two-phase commit also works faster than its simulation, but it is still around 3 times slower that value-sets.

6.7 The Influence of NAT Devices

During the first part of this chapter, we analysed the behaviour of the relaxed ring and Chord with connectivity quality q from 0.9 to 1, where a quality of $q = 1$ imply a perfectly connected network with transitive connectivity. A connectivity of $q = 0.9$ represents an artificially bad scenario for nodes having public IP addresses. However, when we introduce network address translation (NAT) devices, the statistics show that we could have up to 80% of peers behind NATs, making it a quality of $q = 0.36$ as we will explain later. This evaluation is the result of a collaboration with John Ardelius, who compiled the plots presented in this section.

6.7.1 Related Work on the Study of NATs

Understanding how peer-to-peer systems behave on the Internet has received a lot of attention in the recent years. The increase of NAT devices have been an important issue and is one of the main sources for broken network assumptions made in the design and simulation of overlay networks. The studies are mostly related to systems providing file-sharing, voice over IP, video-streaming and video-on-demand. Such systems use overlay topologies different from Chord-like ring, or at most they integrate the ring as one of the components to provide a DHT. Therefore, they do not provide any insight regarding the influence of NAT on ring-based DHTs.

A deep study of Coolstreaming [LQK+08], a large peer-to-peer system for video-streaming, shows that at least 45% of their peers sit behind NAT devices. They are able to run the system despite NATs relying on permanent servers logging successful communication to NATted peers, to be reused in new communication. In general, their architecture rely on servers out of the peer-to-peer network to keep the service running. A similar system, that in addition to streaming provides video-on-demand, is PPLive. Their measurements on May 2008 [HFC+08] indicates 80% of peers behind NATs. The system also uses servers as loggers for NAT-traversal techniques, and the use of DHT is only as a component to help trackers with file distribution.

With respect to file-sharing, a study on the impact of NAT devices on BitTorrent [LP09] shows that peers behind NATs get an unfair participation. They have to contribute more to the system than what they get from it, mainly because they cannot connect to other peers behind NATs. It is the opposite for peers with public IP addresses because they can connect to many more nodes. It is shown that the more peers behind NATs, the more unfair the system is. According to [JOK09], another result related to BitTorrent is that NAT devices are responsible for the poor performance of DHTs as "DNS" for torrents. Such conclusion is shared by apt-p2p [DL09] where peers behind NATs are not allowed to join of the DHT. Apt-p2p is a real application for software distribution used by a small community within Debian/Ubuntu users. It uses a Kademlia-based [MM02] DHT to locate peers hosting software packages. Peers behind NATs, around 50% according to their measurements, can download and upload software, but they are not part of the DHT, because they break it. To appreciate the impact NAT devices are having on the Internet, apart from the more system specific measurements referenced above, we refer to the more complete quantitative measurements done in [DPS09]. Taking geography into account, it is shown that one of the worse scenarios is France, where 93% of nodes are behind NATs. One of the best cases is Italy, with 77%. Another important measurement indicates that 62% of nodes have a time-out in communication of 2 minutes, which is too much for ring-based DHT protocols. Being aware of several NAT-traversal techniques, the problem is still far from being solved. We identify Nylon [KPQS09] as promising recent attempt to incorporate NATs in the system design. Nylon uses reactive hole punching protocol to create paths of relay peers to set-up communication. In their work a combination of four kinds of NATs is considered and they are being able to traverse all of them in simulations and run the system with 90% of peers behind NATs. However, their approach does not consider a complete set of NAT types. The NATCracker [REAH09] makes a classification of 27 types of NATs, where there is a certain amount of combinations which cannot be traversed, even with the techniques of Nylon.

6.7.2 Evaluation Model

We will use a slightly different evaluation model to study the impact of NAT devices in ring-based networks. We still keep the quality of connectivity factor q that we used in the first part of the chapter, but now we compute it in a different way, which we call the *c-model*. In this *c-model*, we consider only two groups of nodes: *open* peers and *NATted* peers. An open peer is a node with public IP address or sitting behind a traversable-NAT, meaning that it can establish a direct link to any other node. A NATted peer is a node behind a NAT that cannot be traversed from another NATted peer, or that it is so costly to traverse that it is not suitable for peer-to-peer protocols. Open peers can talk to NATted peers, but NATted peers cannot talk with each others. In the model, each node has a probability p of being a NATted peer. The connectivity quality q of a network, that is the fraction of available links, is

$$q = 1 - c = 1 - p^2 \tag{6.1}$$

In the rest of the section we will use c to denote the square root of the probability of being a NATted peer, and, the fraction of unavailable links.

6.7.3 Skewed Branches

In the case the amount of NATted peers is not larger than half of the size of the network, it is always possible, in theory, to configure a perfect ring. By placing every NATted peer in between two open peers all links are available to communication. Of course, such situation is very unlikely to happen in a dynamically changing network as branches are created as soon as a NATted peer succeeds another. Even in theory, however, it is impossible to avoid two consecutive NATted peers is if the network contains more NATted than open peers.

In Chord, it is mandatory for the joining peer to be able to talk to its successor to perform maintenance. When a branch is created because two peers p and q cannot communicate, q being the joining peer, both nodes will have the same successor s. A new NATted peer having identifier $k \in]p, q]$ will not be able to join because it cannot communicate with its successor candidate q. Given that, the new peer will get a new identifier and try to *re-join*. If it then receives an identifier $r \in]q, s]$ the peer will join successfully, but in a branch. As more and more NATted peers join the network however, it becomes impossible for a NATted peer to join in ranges $]p, q]$ and $]q, r]$. This situation creates a skewed distribution for the range of available places to join the ring, allowing joining NATted peers to join only closer and closer to the root. Figure 6.15 depicts how the situation evolves with more NATted peers.

As the value of c increases, more NATted peers will be part of the network saturating the branches, shrinking the space for new peers to join. Given that, a peer will need several attempts trying to rejoin the network until finding a key-range where it can connect to the root of a branch. Figure 6.16 shows

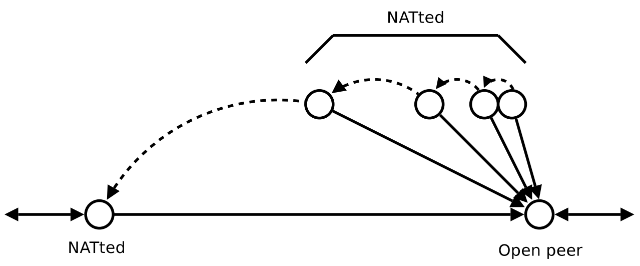


Figure 6.15: Skewed distribution in a branch, shrinking the space for joining.

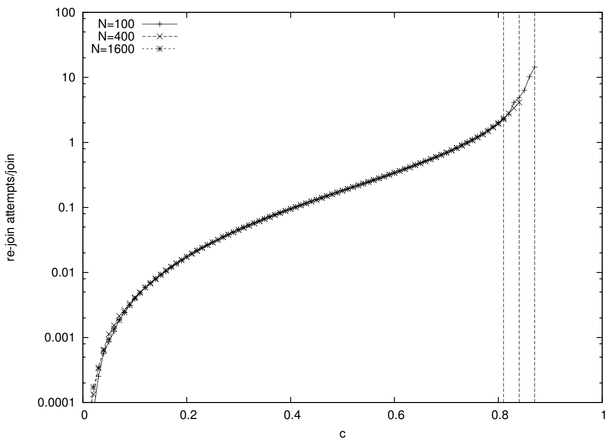


Figure 6.16: Average number of re-join attempts before a node is able to join the ring for different values of c. Vertical lines indicate the point where a some node in the system needs to re-try more than 10,000*N times.

the amounts of re-join attempts as function of c, the variance is presented in Figure 6.17. We can observe a critical value of $c \approx 0.8$ where the amount of attempts is super exponential, meaning basically that no more peers can join the network. This could represent a phase-transition that we will investigate further as future work.

6.7.4 Resilience

Successor List The basic resilience mechanism of ring-based DHTs is the successor list, which contains consecutive peers succeeding the current node. The accuracy of the list is important for efficiency, but it is not crucial for failure recovery. Inaccuracies can be fixed with periodic stabilisation (proactive) as in Chord, or some other mechanism such as correction-on-change (reactive), as we do in the relaxed ring. Resilience is independent of having accurate consecutive

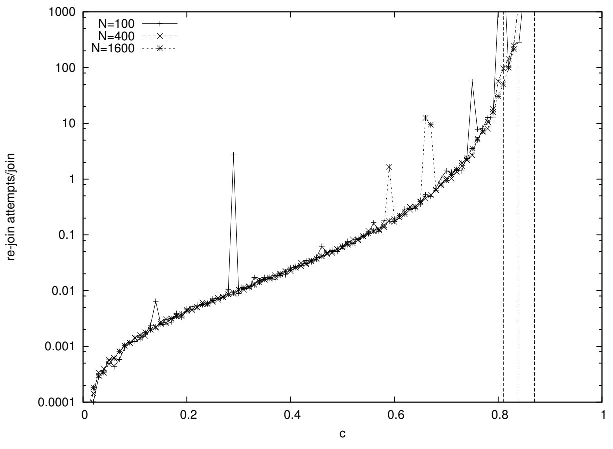


Figure 6.17: The variance of the average number of re-join attempts as function of c.

peers in the successor list and inaccuracy only affects lookup consistency.

The size of the successor list, typically $log(N)$, is the resilience factor of the network. The ring is broken if for one node, all peers in its successor list are dead. In the presence of NATted peers the resilient factor is reduced to $log(N) - n$ for nodes behind a NAT, where n is the amount of NATted peers in the successor list. This is because NATted peers cannot use other NATted peers for failure recovery. The decrease in resilience by the fraction of NATted nodes is possible to cope with for low fractions c. Since there still is a high probability to find another alive peer which does not sit behind a NAT the ring holds together. In fact the NATs can be considered as additional churn and the effective churn rate becomes

$$r_{eff} = r(1 - c) \tag{6.2}$$

For larger values of c, however, the situation becomes intractable. The effective churn rate quickly becomes very high and breaks the ring.

Recovery List As we already mentioned, the resilient factor of the network decreases to $log(N) - n$. To remedy this a first idea to improve resilience is to filter out NATted peers from the successor list. However, the successor list is propagated backwards, and therefore, the predecessor might need some of the peers filtered out by its successor to maintain consistent key ranges. Due to the non-transitivity of the model, if the successor cannot talk to some peers, it does not imply that the predecessor cannot establish connection at all. So the filtering operation is not equivalent to all nodes, this technique would only end up decreasing the average size of the successor list, and thereby,decreasing the resilience factor.

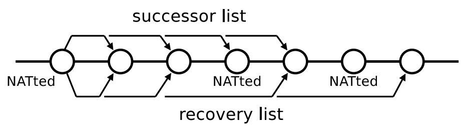


Figure 6.18: Two lists looking ahead: the successor and the recovery list.

We propose to use a second list looking ahead in the ring, denoted the *recovery list*. The idea is that the successor list is used for propagation of accurate information about the order of peers, and the recovery list is used for failure recovery, and it only contains peers that all nodes can communicate with to, that is open peers.

The recovery list is initially constructed by filtering out NATted peers from the successor list. If the size after filtering is less than $log(N)$, the peer requests the successor list of the last peer on the list to keep on constructing the recovery list. Ideally, both lists would be of size $log(N)$. Both lists are propagated backwards as the preceding nodes perform maintenance. Figure 6.18 shows the construction of the recovery list at a NATted peer.

Because both lists are propagated backwards, we have observed that even for open peers is best to filter out NATted peers from their recovery lists even when they can establish connection to them. The reason is that if an open peer keeps references on its recovery list, those values will be propagated backward to a NATted peer who will not be able to use them for recovery, reducing its resilient factor, incrementing its cost of rebuilding a valid recovery list, and therefore, decreasing performance of the whole ring. If no NATted peer is used in any recovery list, the ring is able to survive a much higher degree of churn in comparison to rings only using the successor list for failure recovery.

To study the efficiency of recovery list, we test it with different values of c and different periodic stabilization rate r. The reason to use periodic stabilization as in Chord is that it allows us to study the influence of churn on different ring-based overlay networks. The stabilization rate we use for this evaluation corresponds to the average amount of stabilization rounds a peer performs in its lifetime. A higher stabilization rate means a smaller impact of the churn. Figure 6.19 show the average size of the recovery list as a function of both r and c. In other words, it shows how the recovery list is affected as function of churn and the amount of NATted peers. When there are no NATted peers ($c = 0$), the recovery list is slightly affected by churn, creating no real risk for the network. For higher values of c, it is possible to break the ring by increasing the churn ratio. We can observe the size of the recovery list dropping rapidly before the ring breaks. The higher the c, the lower the churn ratio needed to break the ring.

Permanent Nodes To make the ring resilient to extreme conditions of churn and large amount of NATted peers our study shows that it is necessary to

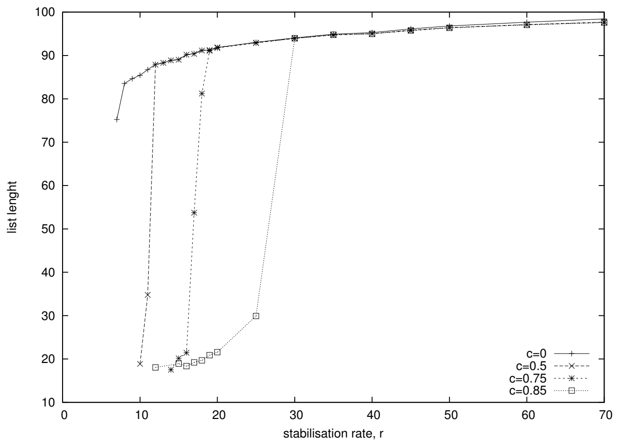


Figure 6.19: Size of the recovery list for various c values as function of stabilisation rate. Before break-up there is a rapid decrease in number of available pointers.

introduce permanent open peers, and keep some of them in the recovery list of each peer. This strategy does not go against the self-organisation property of the system, nor against self-configuration. It only adds predefined resilient information to each peer. It can be seen as a requirement for service providers that want to guarantee the stability of the system.

Figure 6.20 shows a Chord ring with a large amount of NATted peers. We can observe that the core ring contains very few nodes, which are mainly permanent open peers. There are also some NATted peers in the core ring between open peers. As expected, the open peers at the core ring become overloaded for routing, but at least, the ring survives the extreme conditions.

Having permanent nodes is a technique already used by existing peer-to-peer systems. In apt-p2p [DL09], bootstrapping peers running on PlanetLab are used to keep a Kademlia ring alive. Coolstreaming [LQK$^+$08] and PPLive [HFC$^+$08] use logging servers to help NATted peers to establish communication with other peers. These examples indicates that using permanent nodes appear as a feasible approach to keep the service running.

6.8 Conclusion

This chapter has helped us to validate our claims from Chapters 3 and 4. We have shown that the relaxed ring presents much less lookup inconsistencies compared to Chord. This result is observed even when we tested the relaxed ring with worse connectivity quality than Chord. We also observed that correction-on-change provides cost-efficient ring maintenance, specially compared to peri-

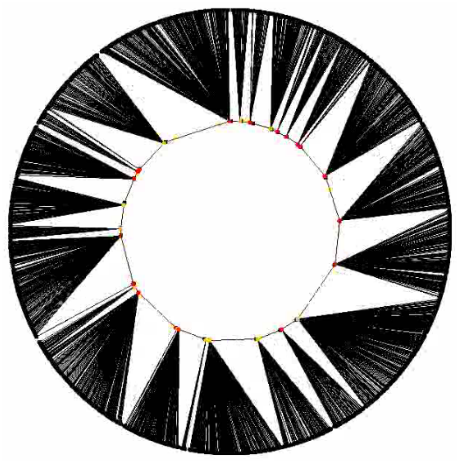


Figure 6.20: The ring survives high churn and large amount of NATted peers. However, hotspots are created in the core ring.

odic stabilization. Relaxing the structure not only improve consistency, but it also reduced maintenance cost.

The price to pay due to the relaxation of the ring is an extra routing cost. Chord achieves $O(logN)$ and the relaxed ring adds the size of branches to that cost. However, experimental results show that average branch's size is less that two peers, and that the impact in the whole network is actually less than 0.5. This means that the cost is negligible, specially if we combine the result with PALTA strategy to provide a self-adaptable finger table. PALTA can be efficient in two directions. It makes a more efficient use of the connectivity in small networks, and it reduces the amount of hops to resolve lookups in larger network. This is because some peers end up with a larger routing table than regular DKS finger table, making more significant jumps to route messages.

We designed and implemented the $c - model$ to understand the influence of NAT devices on the relaxed ring and other ring-based overlay networks. We conclude that Chord-like rings could work only for very low values of c, smaller than 0.05. The relaxed ring handles until $c \approx 0.8$, where branches start collapsing because they shrink the address space, not letting new peers join the network. To improve network's resilience, we identified the need for a recovery list that filters NATted peers to complement the successor list.

With respect to Trappist's transactional protocols, we have quantified the impact in performance of including a set of replicated transaction managers to gain in fault-tolerance. We have also validated the improvement in performance of lock-free transactional protocol, which is the result of relaxing order of elements and versioning in key/value-sets.

Chapter 7

Beernet's Design and Implementation

A beer a day keeps the doctor away

Common sense

Beernet stands for þeer-to-þeer network, where words *peer* and *beer* are mixed. The word peer is used to emphasise the fact that this is a peer-to-peer system. Beer is a known mean to achieve relaxation, and the *relaxed*-ring is the network topology we use to built the system. This chapter describes not only Beernet's implementation, but it also discusses the programming principles and abstractions we have chosen to design a modular system that deals with the intrinsic characteristic of distributed computing. We will review its architecture and design decisions, describing how low-level errors can be hidden from higher-levels. First, we will need to review some general concepts on concurrent and distributed programming to understand the design decision we have taken. This chapter is written not only in the context of self-management and decentralized systems, but also with an interest for programming language abstraction and software engineering.

7.1 Distributed Programming and Failures

The key issue in distributed programming is partial failure. It is what makes distributed programing different from concurrent programming. This unavoidable property causes uncertainty because we cannot know whether a remote entity is ever going to reply to a message. It is also the reason why remote procedure call and remote method invocation (RPC and RMI) are difficult to use. In “A note on Distribution” [WWWK94], four main concerns on distributed programming are discussed: *latency*, *memory access*, *concurrency* and *partial failure*. Latency is not a critical problem because it does not change the semantics of performing an operation on a local or a distributed entity. It just makes things go a bit slower. Memory access is solved by using a virtual machine that abstracts the access, and then it does not change the operational semantics either. A more difficult problem is concurrency. The middleware has to guarantee exclusive access to the state to avoid race conditions. There are different techniques such as data-flow, monitors or locks, that makes possible the synchronization between processes achieving a coherent state. Given that, and even though it is not trivial to write concurrent programs correctly, it is not a critical problem either. What really breaks transparency is partial failure. Basically, distribution transparency works as long as there is no failure.

A partial failure occurs when one component of the distributed system fails and the others continue working. The failure can involve a process or a link connecting processes, and the detection of such a failure is a very difficult task. In distributed environments such as the Internet, it is impossible to build a perfect failure detector because when a process p stops responding, another process p' cannot distinguish if the problem is caused by a failure on the link connecting process p or the crash of the process p itself. This explanation might be trivial, but it is usually forgotten. Failures are a reality on distributed systems, this is why we consider the definition of a distributed system given by Leslie Lamport very accurate:

> “A distributed system is one in which the failure of a computer you did not even know it existed can render your own computer unusable”

Even though this definition does not describe the possibilities of a distributed system, it makes explicit why distributed computing is special. It is very important to know how the system handles the failure of part of the system. We have already discussed this concept in the motivation of this dissertation. We have used the concept along the analysis of the state of the art, and we have use it in the design of the relaxed ring and the transactional layer.

The classical view of distributed computing sees partial failure as an error. For instance, a remote method invocation (RMI) on a failed object raises an *exception*. This approach actually goes against distribution transparency, as it is explained in [GGV05] because the programmer is not supposed to make the distinction between a local and a distributed entity. Therefore, an exception due to a distribution failure is completely unexpected, breaking transparency. Another less fundamental issue but still relevant, is that RMI and RPC are conceived as synchronous communication between distributed processes. Due to network latency, synchronous communication is not able to provide good performance because the execution of the program is suspended until the answer (or an exception) arrives.

New trends in distributed computing, such as ambient intelligence and peer-to-peer networks, see partial failure as an *inherent characteristic* of the system. A disconnection of a process from the system is considered normal behaviour, where the disconnection could be a gentle leave, a crash of the process, or a failure on the link. We believe that this approach leads to more realistic language abstractions to build distributed systems. We believe that the most convenient mechanism to develop peer-to-peer applications effectively is by using *active objects* that communicate via *asynchronous message passing*. These active object are very similar to actors [AMST97] . We also use *fault streams* per distributed entity to perform failure handling. In this chapter we show that this works better than the usual approach of using RMI. We define our peers as lightweight actors and we use them to build a highly dynamic peer-to-peer network that deals well with partial failure and non-transitive connectivity. Our model is influenced by the programming languages Oz [Moz08, VH04] and Erlang [Arm96], and by the algorithms of the book "Introduction to Reliable Distributed Programming" [GR06], which we already introduced in Chapter 3 to describe our algorithms for the relaxed ring. In the following section we describe the model more in detail, focusing also in their component architecture.

7.2 Event-Driven Components

The algorithms for reliable distributed programming presented in [GR06] are designed in terms of components that communicate through events. Every component has its own state, which is encapsulated, and every event is handled in a mutually exclusive way. The model avoids shared-state concurrency because the state of a component is modified by only one event at the time.

Every component provides a specific functionality such as point-to-point communication, failure detection, best effort broadcast, and so forth. Components are organized in layers where the level of the abstraction is organized bottom-up. A higher-level abstraction requests a functionality from a more basic component by triggering an event (sending a request). Once the request is resolved, an indication event is sent back to the abstraction (sending back a reply). Algorithm 17 is taken from the book, where only the syntax has been slightly modified. It implements a best-effort broadcast using a more basic component, (*pp2p*), which provides a perfect point-to-point link to communicate with other processes.

The best-effort broadcast (*beb*) component handles two events: *bebBroadcast* as requested from the upper layer, and *pp2pDeliver* as an indication coming from the lower layer. Every time a component requests *beb* to broadcast a message m, *beb* traverses its list of other peers, triggering the *pp2pSend* event to send the message m to every peer p. Every p is a remote reference, but it is the *pp2p* component which takes care of the distributed communication. At every receiving peer, the *pp2p* component triggers *pp2pDeliver* upon the reception of a message. When *beb* handles this event, it triggers *bebDeliver* to

Algorithm 17 Best Effort Broadcast

```
upon event ⟨ bebBroadcast | m ⟩ do
    for all p in other_peers do
        trigger ⟨ pp2pSend | p, m ⟩
    end
end

upon event ⟨ pp2pDeliver | p, m ⟩ do
    trigger ⟨ bebDeliver | p, m ⟩
end
```

the upper layer, as seen in Algorithm 17. It is important to mention that *beb* does not have to wait for *pp2p* every time it triggers *pp2pSend*, and that *pp2p* does not wait for *beb* or any other component when it triggers *pp2pDeliver*. This asynchronous communication between components means that each component can potentially run in its own independent thread.

Using layers of components allows programmers to deal with issues concerning distribution only at the lowest layers. For instance, the component *beb* is conceived only with the goal of providing a broadcast primitive. The problem of communicating with a remote processes through a point to point communication channel is solved in *pp2p*. If a process *p* crashes while the message is being sent, it does not affect the code of *beb*, thus improving the transparency of the component. There is no need to use something like

try ⟨*send m to p*⟩ **catch** ⟨*failure*⟩

It is the responsibility of *pp2p* to deal with the failure of *p*. It is also possible that *pp2p* triggers the detection of the crash of *p* to the higher level, and then it is up to *beb* to do something with it, for instance, removing *p* from the list of other peers to contact. In such a case, the failure of *p* is considered as part of the normal behaviour of the system, and not as an exception. Even though the code for the maintenance of *other_peers* set is not given [GR06], we can deduce it from the implementation of the other components. In Algorithm 18 the *register* event is a request made from the upper layer, and *crash* is an indication coming from the *pp2p* layer.

Algorithm 18 Best Effort Broadcast extended

```
upon event ⟨ bebRegister | p ⟩ do
    other_peers := other_peers ∪ {p}
end

upon event ⟨ crash | p ⟩ do
    other_peers := other_peers \ {p}
end
```

Even though we advocate defining algorithms using event-driven components using the approach of [GR06], there are some important drawbacks to consider. To compose layers, it is necessary to create a channel, connect the components using the channel, and subscribe them to the events they will handle. We find this approach a bit over sophisticated. It could be simplified by talking directly to a component and using a default listener only when necessary. A related problem concerns the naming convention of events. The name reflects the component implementing the behaviour, making the code less *composable*. For instance, if we want to use a fair-loss point-to-point link (*flp2p*) instead of *pp2p*, we would have to change the *beb* code by replacing *pp2pSend* by *flp2pSend*, and instead of handling *pp2pDeliver* we would have to handle *flp2pDeliver*.

Since the architecture considers components and channels, an alternative and equivalent approach would be to use objects with explicit triggering of events as method invocation, instead of using anonymous channels. Using objects as collaborators, they could be replaced without problems as long as they implement the same interface. In such an approach, both *flp2p* and *pp2p* would handle the event *send* and trigger *deliver*.

The other problem of [GR06] is that there is no explanation of how to transfer a message from one process to the other. The more basic component *flp2p* is only specified in terms of the properties it holds, but it is not implemented. There is no language abstraction to send a message to a remote entity.

7.3 Event-Driven Actors

As we have previously mentioned during this dissertation, we have implemented Beernet in Mozart/Oz [Moz08]. Mozart is an implementation of the Oz language, a multi-paradigm programming language supporting functional, concurrent, object-oriented, logic and constraint programming [VH04]. It offers support for distributed programming with a high degree of transparency. Thanks to the multi-paradigm support of Oz, we were able use more convenient language abstractions for distribution and local computing while building Beernet. In this section we discuss the basic language abstractions that we considered appropriate and necessary to implement event-driven components. In addition, we discuss the abstractions that allowed us to improve the approach towards an event-driven actor model, that we also call active objects.

7.3.1 Threads and Data-Flow Variables

One of the strengths of the Oz language is its concurrency model which is easily extended to distribution. The kernel language is based on procedural statements and single-assignment variables. When a variable is declared, it has no value yet, and when it is bound to a value, it cannot change the value. Attempting to perform an operation that needs the value of such a variable will wait if the variable has no value yet. In a single-threaded program, that situation will block forever. In a multi-threaded program, such a variable is very useful to synchronize threads. We call it a data-flow variable. Oz provides lightweight threads running inside one operating system process with a fair thread scheduler.

The code in Algorithm 19 shows a very simple example of data-flow synchronization. First, we declare variables *Foo* and *Bar* in the main thread of execution. Then, a new thread is created to bind variable *Bar* depending on the value of *Foo*. Since the value of *Foo* is unknown, the '+' operation waits. A second thread is created which binds variable *Foo* to an integer. At this point, the first thread can continue its execution because the value of *Foo* is known.

Algorithm 19 Threads and data-flow synchronization

```
declare Foo Bar
thread Bar = Foo + 1 end
thread Foo = 42 end
```

This synchronization mechanism does not need any lock, semaphore, or monitor, because there is no explicit state, and therefore, no risk for race conditions. The values of *Foo* and *Bar* will be the same for all possible execution orders of the threads. Single-assignment variables are also used in languages such as E [MTSL05] and AmbientTalk [DVM+06, VMG+07], where they are called promises or futures. They are combined with the *when* operator as one of the mechanisms for synchronization.

The execution of a concurrent program working only with single-assignment variables is completely deterministic. While this is an advantage for correctness (race conditions are impossible), it is too restrictive for general-purpose distributed programming. For instance, it is impossible to implement a server talking to two different clients. To overcome this limitation, Oz introduces Ports, which are described in the following section.

7.3.2 Ports and Asynchronous Send

A port is a language entity that receives messages and serializes them into an output stream. After creating a port, one variable is bound to the identity of the port. That variable is used to send asynchronous messages to the port. A second variable is bound to the stream of the port, and it is used to read the messages sent to the port. The stream is just like a list in Lisp or Scheme, a concatenation of a head with a tail, where the tail is another list. The list terminates in an unbound single-assignment variable. Whenever a message is sent to the port, this variable is bound to a dotted pair containing the message and a fresh variable.

Algorithm 20 combines ports with threads. First we declare variables P and S. Then, variable P is bound to a port having S as its receiving stream. A thread is created with a *for-loop* that traverses the whole stream S. If there is no value on the stream, the *for-loop* simply waits. As soon as a message arrives on the stream, it is shown on the output console. A second thread is created to traverse a list of beers (*BeerList*, declared somewhere else), and to send every beer as a message to port P. This is a like a barman communicating with a client. Everybody who knows P can send a message to it, as in the third thread, where the list of sandwiches is being traversed and sent to the same port. Beers will appear on the stream in the same order they are sent. Beers and sandwiches will be merged in the stream of the port depending on the order of arrival, so the order is not deterministic between them.

Algorithm 20 Port and asynchronous message passing

```
declare P S
P = {NewPort S}
thread
   for Msg in S do
      {Show Msg}
   end
end
thread
   for Beer in BeerList do
      {Send P Beer}
   end
end
thread
   for Sdwch in SandwichList do
      {Send P Sdwch}
   end
end
```

The send operation is completely asynchronous. It does not have to wait until the message appears on the stream to continue with the next instruction. The actual message send could therefore take an arbitrary finite time, making it suitable for distributed communication where latency is an issue. With the introduction of ports, it is already possible to build a multi-agent system running in a single process where every agent runs on its own lightweight thread. The non-determinism introduced with ports allows us to work with explicit state, and there is no restriction on the communication between agents.

7.3.3 Going Distributed

Event though full distribution transparency is impossible to achieve because of partial failures, there is some degree of transparency that is feasible and useful. Ports and asynchronous message passing as they are described in the previous section can be used transparently in a distributed system. The semantics of `{Send P Msg}` is exactly the same if P is a port in the same process or in a remote peer. In both cases the operation returns immediately without waiting until the message is handled by the port. If there is a need for synchronization, the message can contain an unbound variable as a future. Then, the sending peer waits for the variable to get a value, which happens when the receiving peer binds the variable. This implies that the variable, and whatever is contained in the message, is transparently sent to the other peer. Variable binding must therefore be transparent.

Algorithm 21 does a ping-pong between two different peers. The first lines of the code represent peer A who sends a ping message to peer B. The message contains an unbound variable *Ack*, which is bound by peer B to the value `pong`. Binding variable *Ack* resumes the *Wait* operator at peer A. Peer B, code below peer A and indented at the right, makes a pattern matching of every received message with pattern `ping(A)`. If that is the case, it binds A to `pong` and continues with the next message. The pattern matching is useful to implement a method dispatcher as we will see in the next section.

Algorithm 21 Ping-Pong

```
\% at Peer A
declare Ack
{Send PeerB ping(Ack)}
{Wait Ack}
{Show "message received"}

                                  \% at Peer B
                                  for Msg in Stream do
                                     case Msg of ping(A) then
                                        A = pong
                                     end
                                  end
```

This sort of transparency is not difficult to achieve, except when a partial failure occurs. An older release of Mozart, version 1.3.0, takes the classical approach to deal with partial failures: it raises an exception whenever an operation is attempted on a broken distributed reference. Most programming languages take the same approach. This approach has two important disadvantages. First of all, it is cumbersome to add **try** ... **catch** instructions whenever an operation is attempted on a remote entity. More fundamentally, exceptions break transparency when reusing code meant for local ports. If a *distribution* exception is raised, it will not be caught because the code was not expecting that sort of exception.

AmbientTalk [DVM+06, VMG+07] adopts a better approach. In ambient-oriented programming, failures due to temporary disconnections are a very common thing, therefore, no exception is raised if a message is sent to a disconnected remote reference. The message is kept until the connection is restored and the message is resent. Otherwise if the connection cannot be fixed after a certain time, it will be garbage collected. Failures are also a common thing in peer-to-peer networks. The normal behaviour of a peer is to leave the network after some time. Therefore, a partial failure should not be considered as an exceptional situation.

A more recent Mozart release, version 1.4.0, does not raise exceptions when distributed references are broken. It simply suspends the operation until the connection is reestablished or the entity is killed. If the operation needs the value of the entity, for instance in a binding, the thread blocks its execution. If a send operation is performed on a broken port, because of its asynchrony, it still returns immediately, but the actual sending of the message is suspended until the connection is reestablished. This failure handling model [CV06, Col07] is based on a *fault stream* that is attached to every distributed entity [MCPV05, KMV06]. An entity can be in three states, *ok*, *tempFail*, or *permFail*. Once it reaches the permanent failure state, it cannot come back to *ok*, so the entity can be killed. If the entity is in temporary failure for too long, it can be explicitly killed by the application and forced to *permFail*. To monitor an entity's fault stream, the idea is to do it in a different thread that does not block and that can take actions over the thread blocking on a failed entity.

7.3.4 Actors

The actor model [AMST97] provides a nice way of organizing concurrent programming, benefiting from encapsulation and polymorphism in analogous fashion to object-oriented programming. We extend the previous language abstractions with Oz *cells* which are containers for mutable state. State is modified with operator ':=', and it can be read with operator '@'. We do not need to add new language abstractions to build our event-driven actors. Without language support, actors are a programming pattern in Oz as is shown in Algorithm 22. Having ports, the cell is not strictly necessary but we use it to facilitate state manipulation. Every actor runs in its own lightweight thread and communicates asynchronously with other actors through ports. Encapsulation of state is achieved with lexical scoping, and exclusive access to state to avoid race conditions is guaranteed by handling only one event/message at a time.

Algorithm 22 is a working implementation of Algorithms 17 and 18 using the language abstractions we have described in this section. It is written in Oz without syntactic support for actors but the semantics are equivalent. The function *NewBestEffortBroadcast* creates a closure containing the state of the actor and its behaviour. The state includes a list of *OtherPeers* and another actor implementing perfect point-to-point communication, which is named *ComLayer* to make explicit that it could be replaced by any actor that understands event *send*, and not only *pp2p*.

The behaviour is implemented as a set of procedures where the signature of the event is specified in each procedure's argument. For instance, the declaration on code line 9 reads that procedure *Receive* implements the behaviour to handle **upon event** *deliver(Src Msg)*. The variable *Listener* represents the actor in the upper layer.

Variable *SelfPort* is bound to the port that will receive all messages coming from other actors. A thread is launched to traverse the *SelfStream*. For every message that arrives on the stream, pattern matching checks the label of the message to invoke the corresponding procedure. This part of the code represents the method dispatching of the actor. In the Beernet implementation, the creation of the port and the method dispatching are modularized to avoid code duplication, thus reducing the code size of every actor.

The book [GR06] contains complementary material including a Java implementation of the *beb* component. Discarding comments and import lines, the implementation takes 67 lines of code, with the component infrastructure already abstracted. It is worth mentioning that a large number of lines are dedicated to catch exceptions. Equivalent functionality within the Beernet actor model takes only 33 lines.

7.4 The Ring and the Problems with RMI

The architecture of Beernet is based on layers that abstract the different concepts involved in the construction of the peer-to-peer network. A closely related work is the Kompics component framework [AH08], which follows the component-channel approach of [GR06] using a similar architecture. The main difference with Beernet is that instead of having components that communicate through channels, we decided to use event-driven actors. We will describe more in detail Beernet's architecture in section 7.6. In this section we give more details about the lower layer concerning the overlay maintenance to spot the differences between the relaxed ring and the Chord ring [SMK+01]. This last one being the starting point of many other SONs.

In previous chapters we have explained how a ring-based DHT works. As a summary, peers are organized clockwise in a ring according to their identifiers, forming a circular address space of N hash keys. Every peer joins the network with an identifier. The identifier is used to find the correct predecessor and successor in the ring. When peer q joins in between peers p and s, it means that $p < q < s$ following the ring clockwise. Peer s accepts q as predecessor because it has a better key than p. Another reason to be a better predecessor, is that the current predecessor is detected to have crashed. Hence, the maintenance of the ring involves *join* and *crash* events, and it must be handled locally by every peer in a decentralized way.

To keep the ring up to date, Chord performs a periodic stabilization that consists in verifying each successor's predecessor. From the viewpoint of the peer performing the stabilization, if the predecessor of my successor has an identifier between my successor and myself, it means that it is a better successor for me and my successor pointer must be updated. Then, I notify my successor. Algorithm 23 is taken from Chord [SMK+01]. Only the syntax is adapted. The

Algorithm 22 Beernet Best Effort Broadcast

```
fun {NewBestEffortBroadcast Listener}
   OtherPeers ComLayer
   SelfPort SelfStream
   proc {Broadcast broadcast(Msg)}
      for Peer in OtherPeers do
         {Send ComLayer send(Peer Msg)}
      end
   end

   proc {Receive deliver(Src Msg)}
      {Send Listener Msg}
   end

   proc {Add register(Peer)}
      OtherPeers := Peer | @OtherPeers
   end

   proc {Crash crash(Peer)}
      OtherPeers := {Remove Peer @OtherPeers}
   end
in
   OtherPeers = {NewCell nil}
   ComLayer = {NewPP2Point SelfPort}
   SelfPort = {NewPort SelfStream}
   thread
      for M in SelfStream do
         case {M.label}
         of broadcast then {Broadcast M}
         [] deliver then {Receive M}
         [] register then {Add M}
         [] crash then {Crash M}
         end
      end
   end
   SelfPort
end
```

big problem with this algorithm is the instruction

```
x := successor.predecessor
```

Asking for successor's predecessor is done using RMI. This means that the whole execution of the component waits until the RMI is resolved. There is no conflict resolution if *successor* is dead or dies while the RMI is taking place. If there is a partial failure, the algorithm is simply broken.

Algorithm 23 Chord's periodic stabilization

```
upon event ⟨ stabilize | ⟩ do
    x := successor.predecessor
    if x ∈ (self, successor) then
        successor := x
    end
    successor.notify(self)
end
upon event ⟨ notify | src ⟩ do
    if predecessor is nil or src ∈ (predecessor, self) then
        predecessor := src
    end
end
```

An improved version of the stabilization protocol is given in Algorithm 24 using event-driven actors. The representation of a peer is a data structure having *Peer.id* as the integer identifying the peer, and *Peer.port* as the remote reference, being actually an Oz port. The '.' is not an operator over an actor or an object. It is just an access to a local data structure. The '. . . ' in the algorithm hide the state declaration and the method dispatcher loop. The '$<$' operator defines the order in the circular address space. We use it here for simplicity without changing the semantics of the algorithm.

Stabilization starts by sending a message to the successor with an unbound variable X to examine its predecessor. The peer then launches a thread to wait for the variable to have a value, and once the binding is resolved, it sends a message to itself to verify the value of the predecessor. This pattern is equivalent to the *when* abstraction in E [MTSL05] and AmbientTalk [VMG$^+$07]. By launching the thread, the peer can continue handling other events without having to wait for the answer of the remote peer. If the remote peer crashes, the *Wait* will simply block forever without affecting the rest of the computation. When the *Wait* continues, the peer sends a message to itself to serialize the access to the state with the handling of other messages. Otherwise there would be a race condition.

Beernet uses a different strategy for ring maintenance as it was explained in Chapter 3. Instead of running a periodic stabilization, it uses a strategy called *correction-on-change.* Peers react immediately when they suspect another peer to have failed. The failed peer is removed from the routing table, and if it happens to be the successor, the peer must contact the next peer to fix the

Algorithm 24 Chord's improved periodic stabilization

```
fun {NewChordPeer Listener}
   ...
   proc {Stab stabilize}
      X
   in
      {Send Succ.port getPredecessor(X))}
      thread
         {Wait X}
         {Send Self.port verifySucc(X)}
      thread
   end
   proc {Verify verifySucc(X)}
      if Self.id < X.id < Succ.id then
         Succ := X
      end
      {Send Succ.port notify(Self))}
   end
   proc {GetPred getPredecessor(X)}
      X = Pred
   end
   proc {Notify notify(Src)}
      if Pred == nil orelse Pred.id < Src.id < Self.id then
         Pred := Src
      end
   end
   ...
end
```

ring. To contact the next successor, every peer manages a successor list, which is constantly updated every time a new peer join or if there is a failure.

Algorithm 25 presents part of a *PBeer* actor, which is a Beernet peer. The algorithm has been already presented in Chapter 3 as language independent. In this section we presented how it is really implemented. Failure recovery works as follows: when peer P fails, a low-level actor running a failure detector triggers the $crash(P)$ event to the upper layer, where *PBeer* handles it. *PBeer* adds the crashed peer to the crashed set and removes it from its successor list. If the crashed peer is the current successor, then the first node from the successor list is chosen as the new successor. A *notify* message is sent to the new successor. When a node is notified by its new predecessor, it behaves as a Chord node, but in addition, it replies with the *updSL* message containing its successor list. In this way, the successor list is constantly being maintained.

Algorithm 25 Beernet's failure recovery

```
fun {NewPBeer Listener}
   ...
   proc {Crash crash(Peer)}
      Crashed := Peer | @Crashed
      SuccList := {Remove Peer @SuccList}
      if P == @Succ then
         Succ := {GetFirst SuccList}
         {Send Succ.port notify(Self)}
      end
   end
   proc {Notify notify(Src)}
      if {Member Pred @Crashed}
      orelse Pred.id < Src.id < Self.id then
         Pred := Src
      end
      {Send Src.port updSL(Self @SuccList)}
   end
   ...
end
```

7.5 Fault Streams for Failure Handling

As described at the end of subsection 7.3.3, we use a *fault stream* associated to every distributed entity to handle failures. An operation performed on a broken entity does not raise any exception, but it blocks until the failure is fixed or the thread is garbage collected. This blocking behaviour is compatible with asynchronous communication with remote entities. In the fault stream model, presented by Collet et al [CV06, Col07], the idea is that the status of a remote entity is monitored in a different thread. The monitoring thread can take decisions about the broken entity, to terminate the blocking thread. For instance, there are language abstractions to kill a broken entity so it can be garbage collected.

Algorithm 26 describes how we use the fault stream in the implementation of Beernet. There is an actor in charge of monitoring distributed entities called *FailureDetector*. Upon event *monitor(Peer)*, the actor uses the system operation *GetFaultStream* to get access to the status of the remote peer. The fault stream is updated automatically by the Mozart system, which sends heartbeat messages to the remote entity to determine its state. When the state changes, the new state appears on the fault stream. If the connection is working, the state is set to *ok*. If the remote entity does not acknowledge a heartbeat, it is suspected of having failed, and therefore, the state is set to *tempFail*. Since Internet failure detectors cannot be strongly accurate, the state can switch between *tempFail* and *ok* indefinitely. As soon as the state is set to *permFail*, however, the entity cannot recover from that state.

If the state is *tempFail* or *permFail*, the actor triggers the event *crash(Peer)* to the *Listener*, which represents the upper layer. If the state switches back to *ok*, the event *alive(Peer)* is triggered. It is up to the upper layer to decide what to do with the peer. In the case of Beernet, this is described in algorithm 25.

Algorithm 26 Fault stream for failure detection

```
fun {FailureDetector Listener}
   ...
   proc {Monitor monitor(Peer)}
      FaultStream = {GetFaultStream Peer}
   in
      for State in FaultStream do
         case State
         of tempFail then {Send Listener crash(Peer)}
         [] permFail then {Send Listener crash(Peer)}
         [] ok then {Send Listener alive(Peer)}
         end
      end
   end
   ...
end
```

7.6 General Architecture

Now that we have reviewed the fundamental concepts that allowed us to implement our components as event-driven actors, we review the general architecture of Beernet. In this section we use the words actors and component indifferently.

We have mentioned that Beernet is globally organized as a set of layers providing higher-level abstractions with a bottom-up approach. Figure 7.1 gives the global picture of how actors are organized. Components with gray background highlights the contribution of this dissertation.

The bottom layer is the *Network* component. This actor is composed by four other actors. The most basic communication is provided by perfect point-to-point link (*Pp2p link*) that simply connects two ports. The *Peer-to-peer link* allows a simpler way of sending messages to a peer using its global representation, instead of extracting the port explicitly every time a message is to be sent. *Peer-to-peer link* uses *Pp2p link*. Any NAT traversal protocol must be implemented here, improving the reliability of both link components. However, if there is a link that cannot be established between two peers, the relaxed ring topology will prevent that it generates a failure at the upper layer.

The network uses two failure detectors: one provided by Mozart, and the other one implemented in Beernet itself. *Mozart's failure detector* is the one described in Algorithm 26, taking advantage of the fault-stream of every distributed entity. *Beernet's failure detector* is built as a self-tuning failure detector that uses its own protocol to change the frequency and timeout values of the keep alive messages. It follows the design discussed in Section 4.4. Both failure detectors are eventually perfect. As a remainder of what we already mentioned in previous chapters, an eventually perfect failure detector is strongly complete, and eventually accurate. Strongly complete means that every peer that crashes will be detected. Accurate means that non alive peer will be suspected as failed. Eventually accurate means that every false suspicion will eventually be corrected.

The relaxed ring component uses the network component to exchange messages between directly connected peers, and to detect failures of any of them. It is one of the main contributions of this dissertation, and it is divided into two main components: the *relaxed ring maintenance* and PALTA *Finger Table*. The relaxed ring maintenance runs the protocols we have described in Chapter 3 and 4. Every incoming message comes from the peer-to-peer link, and crash/alive events comes from failure detectors.

The finger table is in charge of efficiently routing messages that are not sent neither to the successor nor the predecessor of a node. The finger table actor can implement several of the routing strategies we discussed in Chapter 2, as long as it is consistent with the relaxed ring topology. For Beernet, we have decided to implement fingers as they are described in Section 3.9, using PALTA strategy, which combines DKS [AEABH03] fingers with self-adaptable behaviour to improve efficiency. This actor is also in charge of monitoring all messages to provide correction-on-use of the routing table. Algorithm 27 shows that when a message arrives to the relaxed ring maintenance, it first verify is the message is for the self node, for any of the branches (backward) or for its successor. If none of the cases is valid, it delegates the event to PALTA the finger table.

The reliable message-sending layer is implemented on top of the relaxed

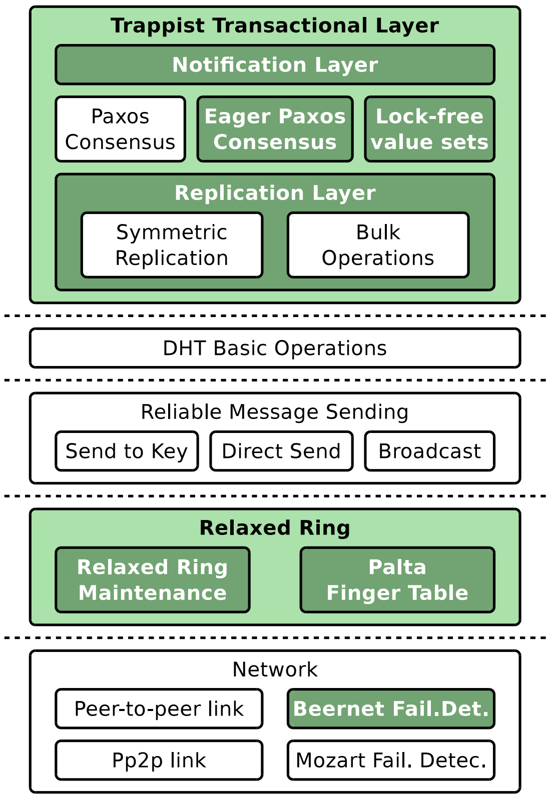


Figure 7.1: Beernet's actor-based architecture. Every component run on its own lightweight thread and they communicate through asynchronous message passing.

Algorithm 27 Messages in the relaxed ring maintenance component are routed to direct neighbours or delegated to the Finger Table

```
proc {Route Event}
  route(msg:Msg src:_ target:Target ...) = Event
in
  if {BelongsTo Target @Pred.id @SelfRef.id} then
    %% This message is for me
    {Send Self Msg}
  elseif {HasFeature Event last} andthen Event.last then
    %% Backward to the branches
    {Backward Event Target}
  elseif {BelongsTo Event.src.id @SelfRef.id @Succ.id} then
    %% I think my successor is the responsible
    {Send @Succ {Record.adjoinAt Event last true}}
  else
    %% Forward the message using the Finger table
    {Send @FingerTable Event}
  end
end
```

ring maintenance. This layer includes the basic services *Reliable Send to a Key*, *Direct Send* to a peer which reference is known, *Multicast* and *Broadcast*. Each of them is running on their own actor, but they can collaborate if necessary, as the relaxed ring maintenance collaborate with the finger table. It is important to identify that direct send is a functionality provided to the upper layers, but it does not necessarily implements a direct link between two peers. If the link between the sender and the target fails, the direct send component will choose a different path to reach the destination. This is how broken links are not propagated as failures to the upper layer. The basic *DHT* with its *put* and *get* operations is implemented on top of the messaging services. None of this components has been originally developed in this dissertation. They are well known results taken from the literature.

The transactional layer *Trappist*, which is presented in detail in Chapter 5, is built on top of the DHT. In Beernet, we have decided to implement the replication layer as part of the transactional layer. This is a major difference with Scalaris [SSR08], because they present their architecture having replication and transaction as two independent layers. We claim that replication needs the transactional mechanism to restore replicas in case of failures. Let us suppose a set of six replicas i, j, k, l, m and n, where the majority i, j, k and l, holds the latest value of the replicated item. If k fails and it is replaced by k', how does k' know which peers hold the latest value of the item? If it only reads from m or n, the majority is broken. If k' reads from all the replica set, it is doing unnecessary work, because it only needs to read from the majority of it. Having the knowledge of the protocol that is used to manage the replica is the best option to keep replica maintenance efficient. That is why in Beernet the replica maintenance also belongs to the transactional layer instead of being an independent layer.

There are still some orthogonal components within the replica management that can be changed by equivalent ones. For instance, we have chosen to work with *Symmetric replication* instead of successor list replication, or leaf set replication. To reach the replicas, the transactional layer will use the *Bulk operations* [Gho06] which can be written for any replication strategy. Since the functionality of these two components are taken from the literature, they are not part of our contribution. Nevertheless, the design of including the replication layer as part of Trappist is part of the results of this dissertation.

The Trappist layer includes four different protocols to provide transactional support: *Two-Phase Commit*, *Paxos Consensus*, *Eager Paxos Consensus*, and *Lock-free value sets*. The four of them are explained in detail in Chapter 5. Note that *Two-phase commit* is not included in Figure 7.1. This is because its use is not recommended for building peer-to-peer applications, due to the lack of a single reliable transaction manager. Its implementation is motivated by purely academic purposes. Our goal was to validate the advantages of Paxos Consensus algorithm, and to be able to compare their performances, as it was shown in Chapter 6. The figure highlights the three components that are part of the contributions of this dissertation.

The protocols Paxos Consensus, Eager Paxos Consensus and Two-Phase commit are used to manipulate key/value pairs. The Lock-free value sets is not just a different transactional protocol, it also provides a different storage abstraction for data collections. Value-sets can be used in combination with the other three protocols. These three protocols use the direct send component to communicate every protocol participant. As we already mentioned, if such link cannot be established, messages are still reliably routed to the destination without triggering errors at this level. This strategy avoids error propagation through the layers. These protocols are enriched by a notification layer that helps in the development of synchronous collaborative applications.

7.7 Discussion

One of the programming principles we respect in Beernet is to *avoid shared-state concurrency*. We achieve this by encapsulating state, by doing asynchronous communication between threads and processes, by using single-assignment variables for data-flow synchronization, and by serializing event handling with a stream (queue) providing exclusive access to the state. The language primitives of lightweight threads and ports are also used in Erlang [Arm96]. Single-assignment variables also appear in E [MTSL05] and AmbientTalk [VMG$^+$07] in the form of promises, and they are meant for synchronization of remote processes instead of lightweight threads.

The actor model presented here through programming patterns is further developed and supported by E and AmbientTalk. There is one important difference related to the use of lightweight threads. Since they are not supported by these two languages, there is basically only one actor running per process. The actor collaborates with a set of passive objects within the same process. Communication with local objects is done with synchronous method invocation. Communication with other actors, and therefore with remote references, is done with asynchronous message passing. This distinction reduces transparency for the programmer because it establishes two types of objects: local and distributed.

In Beernet, we organize the system in terms of actors only, making no distinction in the send operation between a local and a remote port. Transparency is respected by not raising an exception when a remote reference is broken. There is only one kind of entity, an actor, and only one send operation.

As mentioned in the previous section, Kompics [AH08] is closely related because it is also a component framework conceived for the implementation of peer-to-peer networks. Instead of using actors for composition, it uses event-driven components which communicate through channels.

7.8 Conclusion

We have presented examples in this chapter to highlight the importance of partial failure in distributed programming. The fact that failures cannot be avoided has a direct impact on the goal of transparent distribution which cannot be fully achieved. Therefore, it has also an impact on remote method invocation, the most common language abstraction to work with distributed objects. Because of partial failure, it is very difficult to make RMI work correctly. In other words, RMI is considered harmful. Therefore, we have implemented Beernet where communication between processes and components is done with asynchronous message passing.

Even though full transparency in distributed programming cannot be completely achieved, it is important to provide some degree of transparency. We have shown how *port* references and the *send* operation can be used transparently. This is because send works asynchronously and because a broken distributed reference does not raise an exception in Mozart 1.4.0. Instead, a fault stream associated to every remote entity provides monitoring facilities. The fault stream breaks the transparency in a modular way. It is not transparent because failure only appears on remote entities, but it is modular because it does not interfere with the semantics of the operations performed on remote entities.

We have also described the language abstractions we use to implement Beernet. We have chosen an actor model based on lightweight threads, ports, asynchronous message passing, single-assignment variables and lexical scoping. We have reviewed the general structured of Beernet and we have shown how components interact with each other.

Chapter 8

Applications

The development of software applications is important to validate a programming framework. In our case, we are interested in validating the self-managing properties of our structured overlay network, and the transactional protocols for replicated storage. More concretely, we are interested in validating self-organization and self-healing. We want to observe that applications create peers and join the network without managing their location, or simply connect to existing peers in the network. If self-healing is respected, applications should rely on Beernet's failure recovery mechanism, and data should be preserved if the majority of replicas survives. We want to evaluate if applications use Paxos consensus algorithm, Paxos with eager locking, the notification layer and key/value-sets. To cover a wide range of applications, we need synchronous and asynchronous collaborative applications. To validate Beernet as programming framework we also need an application implemented by other developers.

This chapter describes three applications that apply to several of the above mentioned criteria: Sindaca, DeTransDraw, and a decentralized wiki. Sindaca and the wiki are basically asynchronous collaborative applications and basically use Paxos. DeTransDraw benefits from the eager-Locking protocol to provide synchronous collaboration. Key/value-sets are used in Sindaca and DeTransDraw. The decentralized wiki has been implemented by students under the supervision of the author. The fact that the implementation has been done by other developers helps us to validate Beernet as programming framework.

8.1 Sindaca

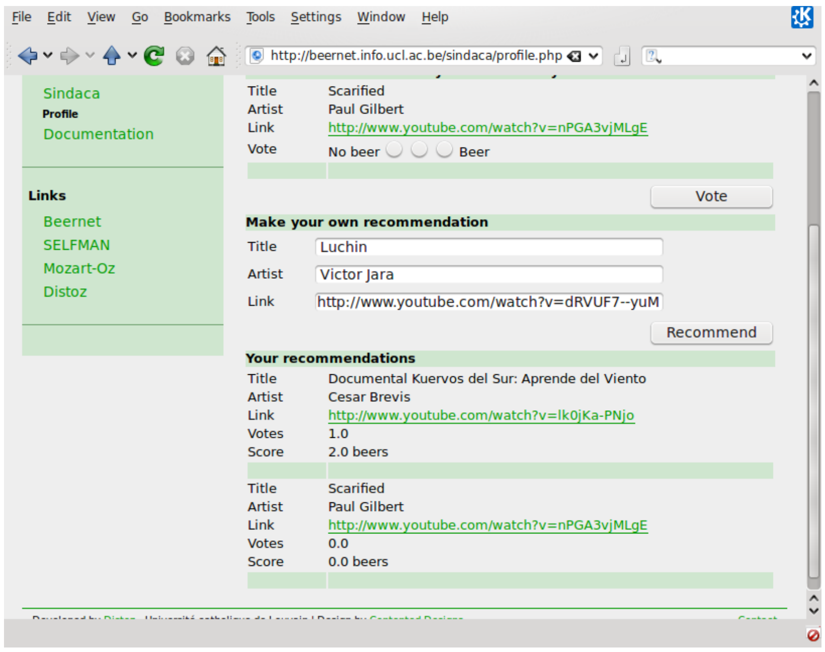
This section presents the design and functionality of our community-driven recommendation system named Sindaca, which stands for Sharing Idols N Discussing About Common Addictions. The name spots the main functionality of this application which is making recommendations on music, videos, text and other cultural expressions. It is not designed for file sharing to avoid legal issues with copyright. It allows users to provide links to official sources of titles. Users get notifications about new suggestions, and they can vote on the suggestions to express their preferences. It is expected that users build communities based on their common taste. The system is implemented on top of Beernet [MCV09, MV10], presented in Chapter 7. The data of the system is symmetrically replicated on the network using the transactional layer for decentralized storage management, Trappist, presented in Chapter 5.

We have implemented a web interface to have access to Sindaca. All requests done through the web interface are transmitted to a peer in the network which triggers the corresponding operations on the peer-to-peer network. The results are transmitted back to the web server, which presents the information in HTML format as in any web page. Using a web interface to transmit information between the end user and the peer-to-peer network has been used previously in various projects. A very related one is the peer-to-peer version of the Wikipedia using Scalaris [PRS07, SSR08]. We have extended this architecture with a notification layer which allows eager information updates. This layer is also used in the DeTransDraw application, as we will see in section 8.2. This eager notification feature is not provided on Sindaca's web interface. However, a client implemented in Mozart [Moz08] can access directly the network and benefit from the feature.

To generalize similitudes and differences between Sindaca and the above mentioned applications, we can say the following: Wikipedia on Scalaris uses *optimistic* transactions using the Paxos consensus algorithm. DeTransDraw uses *pessimistic eager-locking* transactions using Paxos consensus algorithm with a *notification layer*. Sindaca is a combination of those strategies. It uses *optimistic* transactions with Paxos extended with the *notification layer*, both implemented in Trappist.

8.1.1 Functionality

The functionality offered by Sindaca is quite straightforward, and it is depicted in Figure 8.1, which is a screenshot of its web interface. Section "Make your own recommendation" shows a form where users can make their recommendations by providing a title, the artist, and the link to the title. Below the form we find section "Your recommendations", which shows the recommendations the user has done so far, and some statistics about them. The upper part of the screenshot shows recommendations that can be voted by the user. This recommendation list gathers not only recommendations done by other users, but also those done by the user itself. More screenshots, and a deeper description of the functionalities can be seen in Appendix B.



Sindaca
Profile
Documentation

Links

Beernet
SELFMAN
Mozart-Oz
Distoz

Title	Scarified
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=nPGA3vjMLgE
Vote	No beer ○ ○ ○ Beer

Vote

Make your own recommendation

Title	Luchin
Artist	Victor Jara
Link	http://www.youtube.com/watch?v=dRVUF7--yuM

Recommend

Your recommendations

Title	Documental Kuervos del Sur: Aprende del Viento
Artist	Cesar Brevis
Link	http://www.youtube.com/watch?v=lk0jKa-PNjo
Votes	1.0
Score	2.0 beers

Title	Scarified
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=nPGA3vjMLgE
Votes	0.0
Score	0.0 beers

Figure 8.1: Sindaca's web interface.

8.1.2 Design and Implementation

First of all, it is important to remark that Sindaca is not implemented on top of a database supporting SQL queries. Sindaca is implemented on top of a transactional distributed hash table with symmetrically replicated state, which uses key/value pairs and key/value-sets as storage abstractions. Each key/value pair/set is known as an *item*. The information of every user is stored as one item. The *value* of such item is a record with the basic information: user's id, username and password. We have chosen a very minimal record to build the prototype, but the value can potentially store any data such as user's real name, contact information, age, description, etc. The *key* of the item is an *Oz name* [Moz08], which is unique and unforgeable, acting as a capability reference [MS03]. This strategy provides us certain level of security, because only programs that are able to map usernames with their capability can have access to the key, and therefore, access to the item. The username-capability mapping is only available to programs holding the corresponding capability to the mapping table.

The functionality of adding a new recommendation, shown in Figure 8.1, makes it clear that a recommendation belongs to a user. Therefore, every user item contains a list of capabilities which are references to recommendations. The functionality of voting also implies that every user item holds a list of

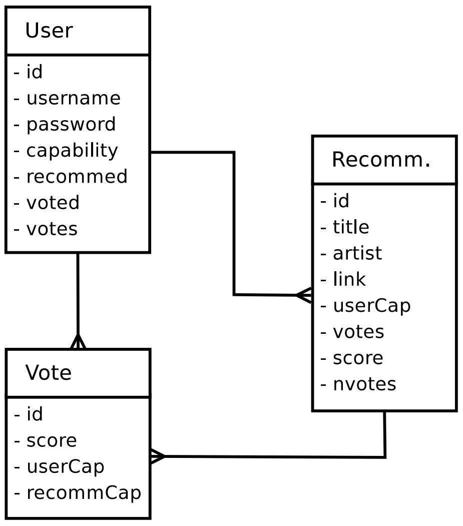


Figure 8.2: Sindaca's relational model.

capability references to votes. The relational model is described in Figure 8.2. We observe that a user can have multiple recommendations and multiple votes. What it is also stored in user's item is the list of recommendations already voted. That list will allows us to filter all other recommendations, presenting to the user only those she still has to vote.

From the relational model we can also observe that every recommendation has a list of votes associated to it. Every vote contains information about the score, the user who made the vote, and the voted recommendation. What it is not shown in the relational model is how to find all the items on the network. There are two other items which store the list of all user's keys and all recommendation keys. Every time a new user or recommendation is created, these global items are modified. There is no global item for votes, because votes are accessible through the users and the recommendations.

Creating a user Code 28 shows the transaction to create a user. The argument `TM` represent the transaction manager created by the peer to execute the transaction. First of all, it is necessary to read the list of users to verify that the new username is not already in use. This is done by reading the item under key `users`, and verifying if `Username` is a member of it. In such case, the transaction is aborted with the operation `{TM abort}`. If the transaction continues, we read the item `nextUser` to get a user identifier. Then, we create a new item with the *capability key* `UserCap`. The value of the new item is a record with the fields we described above, and which follows the relational model on Figure 8.2. Afterwards, the value of the `nextUser` item is incremented, and the item with the list of users is also updated.

As we already mentioned, every user will contribute to the community with a set of recommendations, and it will vote on the recommendation done by others. This information will be stored on three different sets: `recommed`, to stored the title the user has recommended, `votes`, to store the value of each of its votes, and `voted`, to know which recommendations he already voted. These three sets are also identified with a capability, and we will use key/value-set to store them on the DHT.

Algorithm 28 Creating a new user.

```
proc {CreateUser TM}
   Users UserId UserCap
in
   UserCap = {NewName}
   {TM read(users Users)}
   if {IsMember Username Users} then
      {TM abort}
   else
      RecommsCap = {NewName}
      VotesCap = {NewName}
      VotedCap = {NewName}
   in
      {TM read(nextUser UserId)}
      {TM write(UserCap user(username:Username
                             id:UserId
                             passwd:Passwd
                             cap:UserCap
                             recommed:RecommsCap
                             votes:VotesCap
                             voted:VotedCap))}
      {TM newSet(RecommsCap)}
      {TM newSet(VotesCap)}
      {TM newSet(VotedCap)}
      {TM write(nextUser UserId+1)}
      {TM write(users {AddTo Users
                             user(username:Username cap:UserCap})}
      {TM commit}
   end
end
```

As we can observe, this transaction consists of six different access to the hash table, which include adding a new item, and modifying two existing ones. It is very important that all write operations done to the hash table are committed, and if one fails to commit, all of them should abort. Running a distributed transaction is more costly than just using the put/get operations directly on the DHT, but to guarantee the consistency of the data, performing a transaction pays off. Furthermore, even though the operation seems complicated from the point of view of distribution, we can observe that it looks only as a combination of read/write operations that could actually be local ones. The operation *newSet* is a special type of *write*.

Committing a vote The most complex transaction is triggered the voting functionality, and we will explain it using the implementation code as guide. The `PutVote` transaction that performs the needed read/write operations is shown in Code 29. First of all, it is necessary to get the capability to access the user that is voting, and the recommendation that is being voted. These are the variables `UserCap` and `RecommCap`, which belong to the outer scope of `PutVote`. Another variable that belongs to the outer scope is `Vote`, which contains the value of the vote. A new vote item is created with the correspondent vote capability `VoteCap`. This item connects the vote with the user and the recommendation as we will explain now.

Three fields of the voted recommendation are affected. The created vote is added to its list of votes, and the vote counter is increased by one. The score average is recomputed taking the new vote into account. Within each recommendation, we could have used a value-set to store all the votes associated to it. However, this would have add an extra item without improving the performance. The performance is not improved because adding a new vote cannot be done without incrementing the global counter, and the updating the average score, therefore, there is an implicit lock in the addition of each vote. By keeping the set of votes as a list inside the same item, we prevent the information of the recommendation from splitting unnecessarily.

On the other hand, the vote and the recommendation can be associated to the user by simple adding them to the corresponding set. This can be done without modifying the item associated with the core data of the user, and therefore, there is a win by splitting this information into different value-sets.

Note that in this transaction, there is a sequence of read/write operations. Information is taken from an item to modify a related one. In addition, there there are no condition to write the new values, as in Code 28. It is important to clarify that the external functions `Record.adjoin` and `NewScore`, do not perform any distributed operation. They are just for manipulating values and data structures.

Create a recommendation We have seen already in the example of voting that each recommendation is stored in an item having a capability as key. This capability is also used in the users item to identify the recommendations associated to a user. Therefore, to create a new recommendation, the application needs to create a new item for it, increment the global counter of recommendations, and add it to the list of recommendations of the user who created it. We will not include the code sample of putting a recommendation because it follows the pattern described by the previous two examples, and it does not contribute anything new to the understanding of the ease of use of Beernet and

Algorithm 29 Committing a vote on a recommendation

```
proc {PutVote TM}
   User Recomm VoteCap
in
   VoteCap = {NewName}
   {TM write(VoteCap vote(score:Vote
                              recomm:RecommCap
                              user:UserCap))}
   {TM read(RecommCap Recomm)}
   {TM write(RecommCap {Record.adjoin Recomm
                          recomm(score:{NewScore Recomm Vote}
                                 nvotes:Recomm.nvotes+1.0
                                 votes:VoteCap|Recomm.votes)})}
   {TM read(UserCap User)}
   {TM add(User.votes VoteCap)}
   {TM add(User.voted RecommCap)}
   {TM commit}
end
```

its Trappist layer for transactional DHT.

Jalisco transactions The code samples presented in Codes 28 and 29 represent single transactions that will be given to a peer to run it on the network. The outcome of the transaction, either *abort* or *commit*, will be sent to a port where the application will decide the next step. When the transaction to create new users aborts because the username is already in use, the application will need to request the new user to choose a different username before attempting to run a new transaction. In the case of creating new recommendation and voting, getting *abort* as outcome of the transaction only means that there where some concurrent transactions that committed first, creating a temporary conflict with our transaction. In such case, the transaction can be retried without any modification until it is committed. To simplify the process of retrying, we have implemented the procedure *Jalisco*, which comes from the Mexican expression “Jalisco nunca pierde” (*Jalisco never loses*). This procedure will simply retry a transaction until it is committed. The code is shown in Code 30.

The function creates a port to receive the outcome of the transaction. The `InsistingLoop` executes the transaction on the peer `ThePbeer`, and it waits on the stream of the port to check the outcome of the transaction. If it is `abort`, it just continues with the loop. If it is `commit`, it simply returns that the transaction has committed.

This is simply a design pattern to be used in transactional DHTs. It can be seen as a very simply feedback loop. The outcome of the transaction is what is being monitored. The action to be taken in case of `abort` is to insist on running the transaction until the relevant locks are granted. Once they are granted and

Algorithm 30 Jalisco transaction retries a transaction until it is committed

```
fun {Jalisco Trans}
   P S
   proc {InsistingLoop S}
      {ThePbeer runTransaction(Trans P paxos)}
      case S
      of abort|T then
         {InsistingLoop T}
      [] commit|_ then
         commit
      end
   end
in
   {NewPort S P}
   {InsistingLoop S}
end
```

the message `commit` is monitored, the feedback loop ceases to monitor.

8.1.3 Configuration

The current version of Sindaca, which is available for demo testing, is configured with a peer-to-peer network of 42 nodes. All nodes are accessible from the server hosting Sindaca's web page. The server works as one of the possible entry point. The network can also be accessed by contacting directly any peer. Some initial information is stored in the network in order to bootstrap the network and run some tests. This information includes the creation of several users, some recommendations, and some votes too. All the information is stored on the network contacting directly some of the peers, and by running the transactions we have described in this section. Therefore, bootstrapping the information is equivalent to have it entered through the web interface. The current state of development is a proof-of-concept, hence, it is not stable as a rock. With respect to technical details, we have a running Mozart [Moz08] process that listens to the Apache [The09a]-PHP [The09c] service which is reading web requests. This Mozart process connects to a peer in the network in order to trigger the corresponding transaction. Results of the transactions are sent back to the web interface to the Mozart process, communicating with the PHP service.

8.2 DeTransDraw

DeTransDraw is a decentralized collaborative vector-based graphical editor with a shared drawing area. It provides synchronous collaboration between users with graphical support for notifications about other users' activities. Conflict resolution is achieved with a decentralized transactional service with storage replication, and self-management replication for fault-tolerance. The transactional service also allows the application to prevent performance degradation due to network latency, which is an important feature for synchronous collaboration.

8.2.1 TransDraw

DeTransDraw is a redesign of TransDraw [Gro98], a collaborative drawing tool based on a client-server architecture. We first describe TransDraw to identify its advantages and weakness, and then we explain how DeTransDraw can overcome the problems of its centralized predecessor. TransDraw has a shared drawing area where all users has access to all figures of the drawing. The main contribution of TransDraw is the introduction of transactions to manage conflict resolution between users, and to reduce the problems of network latency. The goal is that a user can manipulate the figures immediately, without waiting for the confirmation of a distributed operation. The transaction manager will solve the conflicts afterwards. A transaction is done in two steps: getting the lock and committing. When the user starts modifying one or more figures it request the corresponding locks. When the user finishes the updates, it performs a commit of the transaction with the new value. However, it is possible that the user loses its modification if another user concurrently modifies the figure first.

Figure 8.3(a) describes the protocol where successful modifications done by two users. Users are represented by `client 1` and `client 2`, which are connected to the `Server`. The server stores the full state of the drawing, and manages the locks of every figure. The server is also the transaction manager of all transactions triggered by the users. The protocol shows that `client 1` starts to work on an object of the drawing and it request its lock. Since the lock is not already taken, the `Server` grants it with a `confirm` message. We can see the bar representing the work of `client 1`. While the bar is gray, it is modifying the figure without knowing of the lock will be granted. The *work-bar* becomes green when confirmation arrives, and it is finished when modifications are done, which results in triggering a `commit` message to the server, including the new value of the drawing object. Note that this message also means that locks are released. Two notifications are sent from `Server` to `client 2`: `locked` and `update`. The locked message prevent `client 2` from modifying that particular object. This is what the red bar represents. Was the new value is committed, the update also reaches `client 2` allowing it for requesting the lock, as it is done as its next step. This new lock is now informed to `client 1`.

Figure 8.3(b) shows the same protocol, with the addition of a failed request from `client 2` to acquire the same locked obtained by `client 1`. What we observe is that `client 2` begins to work on the drawing object resulting on

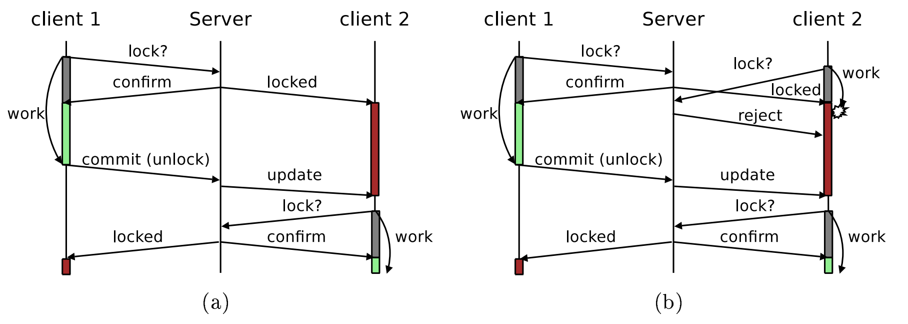


Figure 8.3: Transdraw coordination protocol. In (a) client 1 contacts the server to get the lock and update a figure. Client 2 does another update afterwards. In (b) client 2 gets a rejection to its first attempt of acquiring the lock. When figure's lock is released, the client succeeds getting the lock.

requesting the lock. Its *work-bar* turns from gray to red when the notification of the locked granted to `client 1` arrives. If the notification would have not been sent, the `rejection` notification would have had the same effect. In this case we see that `client 2` has lost some of its work. Note that it is better to be notified earlier, because otherwise the transaction would have failed at the end, when all the work was done. This is the main difference between asynchronous and synchronous collaboration. Asynchronous communication can take an optimistic approach, whereas synchronous collaboration work better with a pessimistic approach, because it is more likely to have a conflict with another user. TransDraw actually merges optimistic and pessimistic approaches. It is optimistic because users can start working immediately even if they do not get the lock. This is essentially to minimize problems associated with latency. And it is pessimistic because it first tries to get the locks, and then it tries to commit. This is to provide a better collaboration between synchronous participants.

8.2.2 TransDraw Weakness

Due to its centralized architecture, TransDraw's main weakness is its single point of failure. The server holds the whole state of the application. Other problems are congestion and scalability. Both of them having the same source we just mentioned. The server is the only transaction manager, and therefore, it is a bottle-neck for all the traffic between users. It is a single point of congestion and it does not scale beyond the capacity of the server.

A disputable issue concerns distributed locks. We will not discuss in details how these problems are solved, but we will briefly describe some possibilities to overcome the issues with locks. First of all, the application has decided to keep strong consistency on the state of the drawing, so it is very difficult to come

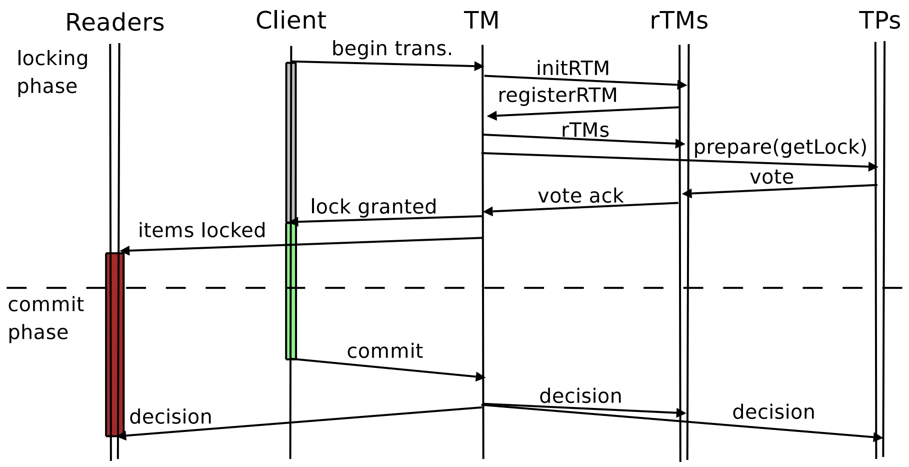


Figure 8.4: DeTransDraw coordination protocol. It combines optimistic and pessimistic approach, using Trappist's eager locking Paxos and the notification layer to propagate the information to the registered readers.

up with a lock-free design. For the case that a client holds a lock for too long, there is a protocol to explicitly request the client to release the lock. In case the client fails without releasing the lock, the transaction manager can release the lock based on failure detection or time-leasing, to prevent problems with false suspicions of failures. In conclusion, there are workarounds to minimize the problems with distributed locks, but the main issue with TransDraw is scalability and fault-tolerance. The server has a limited amount of clients that can handle, therefore, it is not scalable. In addition, the server represents a single point of failure, which the application cannot tolerate. If the server fails, the whole drawing application disappears. This is why a decentralized approach appears as the way to go.

8.2.3 Decentralized TransDraw

We have already discussed the main features and weakness of TransDraw. The aim of DeTransDraw is to provide the same functionality but it removes the server from the design, building the application on top of a peer-to-peer network, making the system fully decentralized. Each transaction runs with its own transaction manager, and the state of the application is spread across the network being symmetrically replicated. This provides not only load balancing but also scalable and fault tolerance.

DeTransDraw is implemented on top of Beernet, and it uses the eager paxos consensus algorithm provided by the transaction layer Trappist. Since Beernet provides a DHT, the drawing information has to be stored in form of items. Each drawing object is an item where its identifier is the key, and the value

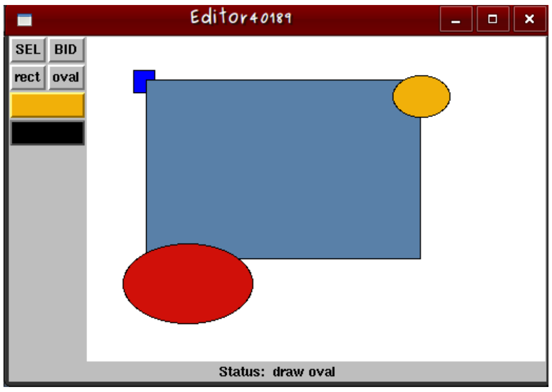


Figure 8.5: DeTransDraw graphical user interface.

corresponds to the position, shape, colour, and other properties of the figure. The application has been implemented in our research group in collaboration mainly with Jérémie Melchior.

Figure 8.4 shows the protocol we described for TransDraw in Section 8.2.1, but the client contacts a transaction manager (TM) instead of a server. In other words, the server is replaced by the peer-to-peer network. The protocol is an instance of Eager Paxos consensus algorithm, as it is described in Section 5.3, combined with the notification layer that communicates with the readers. In this case, the readers are all the other users of DeTransDraw. We can observe that the `client` performs the same operations as in the protocol of Figure 8.3 but with some different names. Requesting the lock is actually `begin transaction`. Confirmation of acquiring the locks is `locked granted`. The `commit` message is the same in both cases. As we mentioned already, the TM is different for every transaction, and the set of replicated TMs is chosen with the same strategy as symmetric replication. The key to generate the replica set is the one of the TM. The transaction participants (TPs) are all the peers storing a replica of the drawing objects involved in the transaction. Therefore, two concurrent transactions modifying disjoint sets of drawing objects could have completely different sets of TM, rTMs and TPs.

We discuss now the graphical user interface of DeTransDraw. Figure 8.5 shows the drawing editor being run by a client. The editor consists of three parts: the canvas, which is the shared drawing area, the toolbar, and the status bar. The state of these last two parts are different on every user depending on their actions. In the toolbar, button `SEL` stands for the selection of an object. Multiple object selection is done by holding the `Shift` key while selecting the objects. The buttons `rect` and `oval` allows the user to draw rectangles and ovals. These are the only figures provided on the first version of DeTransDraw. The two colored buttons represent, from top to bottom, the color of the object and its border. The status bar notifies the user of the action he is currently

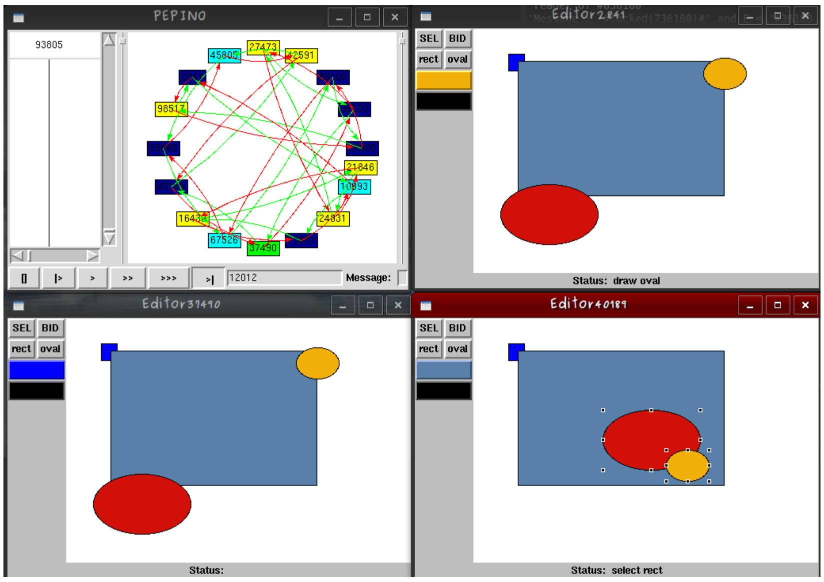


Figure 8.6: Locking phase in DeTransDraw. The user with highlighted window has selected two figures to move them on the drawing. Blue peers on the ring show where the locked replicas are.

doing. In the case of the example, the user has clicked on the `oval` button, so it can draw a yellow oval with black border, as it is described by the coloured buttons. If the user is in selection mode, he is able to select either rectangles or ovals. A selected object appears with eight dots surrounding the object, as it will shown on Figure 8.6.

Figure 8.6 shows how the action of selecting drawing objects changes the state of the network. The figure shows four application windows. The window at the top left corner is a screenshot of PEPINO [GMV07], an application that monitors the network and shows it state. In this case, the network is composed by 17 peers. The other three windows are instances of DeTransDraw which are connected to the network. Looking at the tool bars, we can deduce that the user at the top right corner draw the yellow oval, the user at the bottom left draw the small blue square, and the highlighted user at the bottom right corner draw the blue-gray rectangle. This highlighted user has selected the two ovals acquiring the correspondent locks. We observe in PEPINO some peers in blue, and some other in cyan. The peers in blue are the transaction participants which are currently locked. They are the replicas storing the state of the two ovals. Peers in cyan are the replicated transaction manager, being the peer in green the transaction manager for this operation. The other users do not see the modification of the position of the figures, because the other user has not committed yet its modification.

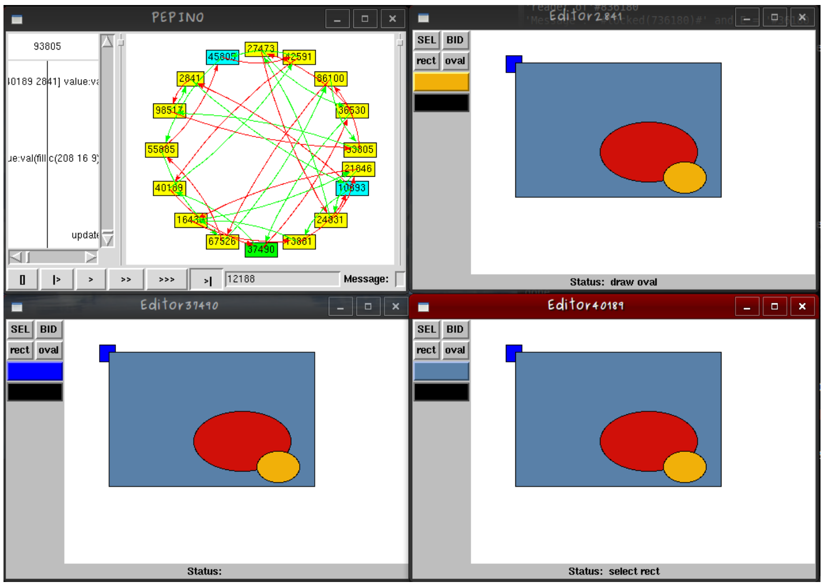


Figure 8.7: Commit phase in DeTransDraw. The user commits the changes, the new state is propagated to the other users, and locks are released.

We observe in Figure 8.7 that locks are released, and the new state of the ovals is replicated. All three instances of DeTransDraw observe the new state and the small black dots of selection disappear from the ovals that were modified. Looking at the network, there are no more blue peers, meaning that locks are released, but there still remains the information about the transaction manager and some of its replicas.

The use of key/value-sets in DeTransDraw is quite straightforward with a clear advantage with respect to key/value pairs. By design, the canvas contains a set of figures. In the DHT, the canvas is stored as a set of keys, where each key represent the item storing the information of each figure. We have observed that it is important to lock a figure to modify it and not to lose the modification at the latest moment. If there is a conflict, work is aborted as early as possible. However, creating a figure should not lock the whole canvas to add one element. If the canvas is represented as a key/value pair, it is necessary to lock it to a figure. Therefore, it is much better to use a key/value-set for the canvas, which is lock free for addition and removal of values.

The software still needs more development to become a real drawing tool, but it is well advanced as a proof-of-concept concerning its decentralized behaviour. It provides the same advantages as TransDraw minimizing the impact of network latency, allowing collaborative work with conflict resolution achieved with transactional protocols. It does not have any single point of congestion or failure, because every transaction has its own transaction manager, with a set of replicated transaction managers symmetrically distributed through the network. State is also decentralized on the DHT, having each item symmetrically replicated. Each transaction guarantees atomic updates of the majority of the replicas.

Instructions to download, install and run DeTransDraw can be found on the web site `http://beernet.info.ucl.ac.be/detransdraw`. There is also a client for Android mobile devices developed mainly by Yves Jaradin with collaboration of the author. Currently, mobile phones can only connect to existing peer-to-peer networks, run the GUI client and use an existing peer to transmit the messages to the network. The mobile phone is no yet a peer on its own because of performance issues. The development of the graphics has to be done in Java, and therefore, interacting Java with Mozart-Oz makes the application a bit slow. These implementation problems are only associated to the graphical part, and they do not represent a problem at the level of peer-to-peer protocols.

8.3 Decentralized Wikipedia

Wikipedia [Wik09] is an online encyclopedia written collaboratively by volunteers, reaching currently more than 13 million articles. A large community of users constantly updates the articles and create new ones. Such system can certainly benefit from scalable storage and atomic commit, being a good case study for self-organizing peer-to-peer networks with transactional DHT. A fully decentralized Wikipedia [PRS07] was successfully built with Scalaris [SSR08], which is based on Chord$^{\#}$ [SSR07] using a transactional layer implementing Paxos consensus algorithm [MH07]. We presented the main characteristics of Chord$^{\#}$ in Chapter 2, and we described in detail Paxos consensus algorithm in Section 5.2. The real Wikipedia runs on a server farm with a fix amount of nodes, with a centrally-managed database. The decentralized version allows the network to add more nodes to the system when more storage capacity is needed. The stored items are symmetrically replicated, and each transaction runs its own instance of a transaction manager, preventing the system from having a single point of congestion.

To validate our implementation of the atomic transactional DHT using Paxos consensus algorithm, which is part of Trappist, running on top of the relaxed ring, we decided to give the task of implementing a decentralized Wikipedia to the students of the course “Languages and Algorithms for Distributed Applications” [Van09], given at the Université catholique de Louvain, as a course for engineering and master students. The students had two weeks to develop their program having access to Beernet's API for building their peer-to-peer network, and for using the transactional layer to store and retrieve data from the network.

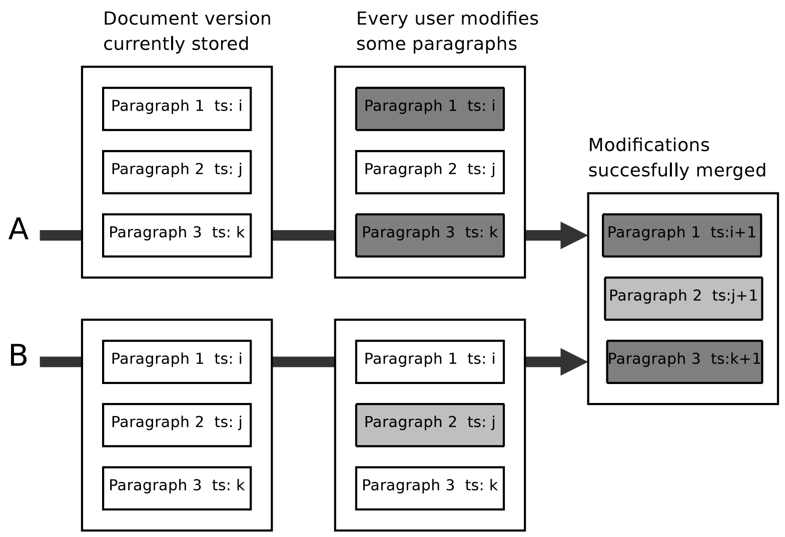


Figure 8.8: Users *A* and *B* modify different paragraphs of the same document. Both can successfully commit their changes because there are no conflicts.

8.3.1 Storage Design for Articles

To store data in a DHT, the information has to be stored as items with a key-value pair. A paragraph in an article was the granularity used to organize the information of the wiki. Articles were stored as a list of paragraphs. Using articles as the minimal granularity would have not been convenient because users never update more than one article at the time. Therefore, the transactional layer would have been used to update only an item at the time, being useful only for managing replica consistency. Furthermore, such granularity would not allow concurrent user to work on the same article. Figure 8.8 depicts how using paragraphs as the minimal granularity can be useful to allow concurrent users updating the same article. On the figure, both users get a copy of an article composed by three paragraphs. Each paragraph has its own version, marked as timestamps (ts). User *A* modifies paragraph 1 and 3, while user *B* modifies paragraph 2. When user *A* commits her changes, the transactional layer guarantees that both paragraph will be updated, or none of them will. This property is particularly interesting if we consider that the article could be source code of a program instead. Allowing only one change could introduce an error in the program. Continuing with the example, since modifications of users *A* and *B* do not conflict, both transactions commit successfully. Consequently, if user *B* would have also modified either paragraph 1 or 3, only one of the commits would have succeeded. It is up to the application to decide how to resolve the conflict.

8.3.2 Reading the Information

The code samples used in this section are taken and modified from one of the student projects, which was called WikiPi2Pedia, with permission of the authors Alexandre Bultot and Laurent Herbin. Getting a copy of an article was divided into two transactions. The first one, wrapped inside the function `GetArticle`, return the list of keys representing the paragraphs associated to a given article, see Code 31. The title of the article is given as the key of the item. The variable `Node` represents the peer, and the operation performed is `runTransaction(Trans Client paxos)` with the following parameters: `Trans` is a procedure receiving a transactional manager `TM` as parameter, which is actually the one over which the operation `read` is performed. The global variable `Client` its a port where the outcome of the transaction, either commit or abort, will be sent. The argument `paxos` is given to chose the protocol to be used for this transaction. Note that key/value-sets cannot be used to store the paragraphs of an article because they have an order between them. Therefore, a key/value pair is used to store the list of paragraphs.

Algorithm 31 Getting the list of paragraphs keys from an article

```
fun {GetArticle Title}
   Value
   proc {Trans TM}
      {TM read(Title Value)}
   end
in
   {Node runTransaction(Trans Client paxos)}
   Value
end
```

The second step for getting the text of an article is to retrieve the values of all the paragraphs. This is done in a similar way in Code 32, with the main difference that many items are read on this transaction. Every resulting `Value` from the `read` operation is added to the list of paragraphs, as it is shown in the following sample code. The operator '|' is used to put the `Value` at the head of the existing list of `Paragraphs`.

Reading an article is divided on these two steps to separate the issue of knowing is an article exist or not. If the article does not exist, a failed value will be the result of the transaction. The disadvantage is that the list of paragraphs can change in between these two steps, and therefore, the displayed article could miss some recently new paragraphs, or still display some deleted information. However, the risk that some other user makes these updates during the session of reading the article is even higher. So the disadvantage can be neglected.

Algorithm 32 Get the text from each paragraph

```
proc {GetPars ParIds}
   Paragraphs = {NewCell nil}
   Trans = proc {$ Obj}
           for K in ParIds do
             Value in
             {Obj read(K Value)}
             Paragraphs := Value|@Paragraphs
           end
         end
in
   {Node runTransaction(Trans Client paxos)}
   @Paragraphs
end
```

8.3.3 Updating Articles

Code 33 performs several transactions to update the article. The modifications are divided into two lists of paragraphs, which are determined by the application: `ToCommit`, containing all paragraphs with modifications, and newly added paragraphs too; `ToDelete` are obviously the paragraphs that will be deleted. These procedures imply several calls to `write` and `remove` on the transactional object. Calling `runTransaction` on the `Node` guarantees that all of them will be committed, or the whole update fails. This version is slightly simplified, because adding and removing items has also implications on the list of paragraphs of the article. The representation of such list is application dependent, so we will not include it on these code samples.

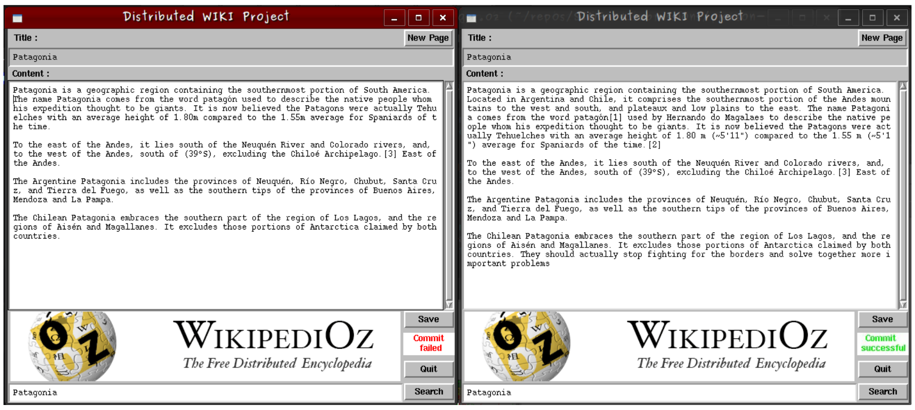
As we can see, reading an article and committing the correspondent updates is fairly simple using the transactional DHT API. As an average, the student projects were about 600 lines of code, including the graphical interface, and the code for bootstrapping the peer-to-peer network. The students were not asked to implement an HTML interface. Instead, they could implement a simple GUI using the Mozart programming system [Moz08], to make it simpler to interact with Beernet. Figure 8.9 is a screenshot of another submitted project called WikipediOz's, with permission of the authors Quentin Pirmez and Laurent Pierson. The figure depicts how the GUI works, and opposite to Figure 8.8, it represents an example of a failed transaction due to a conflict on the edition.

The user running the window at the left of the image has modified paragraph 1 of the article entitled Patagonia. The user running the window on the right has also modified paragraph 1 of the same article, in addition to modifications on paragraph 4. Even without reading the text[1], we can observe that paragraphs 1 and 4 are longer on the right side of the screenshot. By clicking

[1]It is possible to read the text zooming in the pdf version of this dissertation. Text taken from Wikipedia in October 2009 `http://en.wikipedia.org/wiki/Patagonia`

Algorithm 33 Committing updates and removing paragraphs

```
proc {RobustCommit ToCommit ToDelete}
   Trans = proc {$ Obj}
              for UpdPar in ToCommit do
                 {Obj write(UpdPar.id UpdPar.text)}
              end
              for DelPar in ToDelete do
                 {Obj remove(DelPar.id)}
              end
              {Obj commit}
           end
in
   {Node runTransaction(Trans Client paxos)}
end
```



Patagonia

Patagonia is a geographic region containing the southernmost portion of South America.
The name Patagonia comes from the word patagón used to describe the native people whom
his expedition thought to be giants. It is now believed the Patagons were actually Tehu
elches with an average height of 1.80m compared to the 1.55m average for Spaniards of t
he time.

To the east of the Andes, it lies south of the Neuquén River and Colorado rivers, and,
to the west of the Andes, south of (39°S), excluding the Chiloé Archipelago.[3] East of
the Andes.

The Argentine Patagonia includes the provinces of Neuquén, Río Negro, Chubut, Santa Cru
z, and Tierra del Fuego, as well as the southern tips of the provinces of Buenos Aires,
Mendoza and La Pampa.

The Chilean Patagonia embraces the southern part of the region of Los Lagos, and the re
gions of Aisén and Magallanes. It excludes those portions of Antarctica claimed by both
countries.

Patagonia

Patagonia

Patagonia is a geographic region containing the southernmost portion of South America.
Located in Argentina and Chile, it comprises the southernmost portion of the Andes moun
tains to the west and south, and plateaux and low plains to the east. The name Patagoni
a comes from the word patagón[1] used by Hernando do Magalaes to describe the native pe
ople whom his expedition thought to be giants. It is now believed the Patagons were act
ually Tehuelches with an average height of 1.80 m (~5'11") compared to the 1.55 m (~5'1
") average for Spaniards of the time.[2]

To the east of the Andes, it lies south of the Neuquén River and Colorado rivers, and,
to the west of the Andes, south of (39°S), excluding the Chiloé Archipelago.[3] East of
the Andes.

The Argentine Patagonia includes the provinces of Neuquén, Río Negro, Chubut, Santa Cru
z, and Tierra del Fuego, as well as the southern tips of the provinces of Buenos Aires,
Mendoza and La Pampa.

The Chilean Patagonia embraces the southern part of the region of Los Lagos, and the re
gions of Aisén and Magallanes. It excludes those portions of Antarctica claimed by both
countries. They should actually stop fighting for the borders and solve together more i
mportant problems

Patagonia

Figure 8.9: The user at the left modifies paragraph 1 of the article, but the commit fails because the user at the right just committed modifications on paragraphs 1 and 4. Note that the size of paragraphs 1 and 4 is larger at the right window.

on button `Save`, the commit transaction is triggered by the user on the right side, receiving a message *Commit successful*. The user on the left, executing the transaction afterwards, gets an error message *Commit failed*. To complete the description of the screenshot, at the bottom of the window there is a text field that allows searching for articles. The `Search` action performs the reading transactions. At the op of the window there is a button that allows to create new articles.

The feedback from the students helped us to improve our system, and it confirmed us that the provided API is suitable for other programmers to develop applications on top of our system. The students agreed that all the complexity of building the network, routing messages, storing and retrieving data from the replicas, was well hidden behind the API. Unfortunately, they got the feeling that their student project did not let them test their skills on distributed programming for decentralized systems, because they were working on a higher level. This is of course positive for Beernet as programming framework, but we need to reconsider the project as an academic activity.

8.4 Validation Summary

After reviewing the design, functionality and implementation of DeTransDraw Sindacaand the small decentralized wiki, we discuss now Beernet's properties and functionalities used by these applications. Table 8.1 summarizes the relationship between the applications and Beernet. We can observe that all three applications benefit from the self-managing properties of Beernet. None of them needed to do anything concerning the location of peers, or the way they communicated between them. With respect to self-healing, the only action that the applications needed to take was to connect to another peer when their access point failed. The failure recovery at the level of the relaxed ring was transparent to the applications respecting the self-healing property.

When it comes to the use of Trappist's protocols, we observe that Sindaca and the small decentralized wiki used Paxos consensus algorithm, whereas DeTransDraw needed to use Paxos with eager locking and the notification layer. This difference correlates with the kind of application. Both Sindaca and Small wiki are asynchronous applications, whereas DeTransDraw is designed as a synchronous collaborative drawing tool. We can also observed that using key/value-sets is orthogonal to whether Paxos is used with optimistic or pessimistic locking. These data collections are used by Sindaca for the storage of recommendation sets and recommendation voting. In DeTransDraw, they are used by the canvas to store the set of figures. The small decentralized wiki requires a total order of paragraphs in each article, and therefore, it is limited to the use of key/value pairs with versions. Finally, the small wiki meets the requirement of having an application implemented by developers other than Beernet's developers, validating Beernet as programming framework.

Table 8.1: Applications and their use of Beernet's functionality.

	Sindaca	DeTransDraw	Small Wiki
Self-organization	✓	✓	✓
Self-healing	✓	✓	✓
Paxos consensus	✓		✓
Eager locking		✓	
Notifications		✓	
Key/value-sets	✓	✓	
Synchronous collab.		✓	
Asynchronous collab.	✓		✓
Third party develop.			✓

8.5 Conclusion

We have presented three applications on this chapter that make use of Beernet: Sindaca, DeTransDraw and a decentralized wiki. We have described their design, functionality and implementation, and we have evaluated their use of Beernet's properties and functionalities. We have chosen synchronous and asynchronous collaborative systems to cover a wide range of applications. Our evaluation shows a correlation between the use of Paxos with eager locking and the notification layer with synchronous applications, showing the usefulness of these protocols. We have also presented a decentralized wiki application developed by students that were not involved at all in the development of Beernet. This application helps us to validate Beernet as programming framework. The self-managing properties appear to be complete transparent at the level of applications, which validates Beernet's architecture design.

Chapter 9

Conclusions

The ending is just a beginner
the closer you get to the meaning
the sooner you'll know that you're dreaming

"Heaven and Hell" - Black Sabbath

We have started this dissertation discussing about the complexity of building scalable distributed systems. Even when building scalable systems is a high-level design problem, we have identified important issues in the foundations of distributed programming that affect the high-level design decisions. Derived from the inherent asynchrony of distributed systems, dealing with non-transitive connectivity and inaccurate failure detection is unavoidable. The major impact created by these and other issues is that some system's requirements become to hard to meet. Trying to fulfill these requirements can be counterproductive when the incurred cost is larger than the benefit. We propose to relax those requirements to cope with the inherent asynchrony of distributed systems without sacrificing functionality. Applying systematically the relaxation when is needed becomes a design philosophy that we call the relaxed approach. Together with relaxing the requirements, one way of providing the same functionality is by increasing the decentralization and self-managing behaviour of the system. Then, to be able to build applications on top of such systems, we also need to provide robust storage where data can be accessed and modified in a transactional way. Therefore, we want to build scalable systems with two properties: self-managing behaviour and transactional robust storage. In this chapter we summarize how we validate the relaxed approach and how achieved the desired properties for scalable systems.

9.1 Self-Managing Decentralized System

We reviewed existing solutions to scalable systems Chapter 2, where we identified the advantages and disadvantages of each of them. In Chapters 3 and 4, we have presented the relaxed ring topology for self-healing and self-organizing peer-to-peer networks. The topology is the result of relaxing the ring structure introduced by Chord, which requires a perfect ring and transitive connectivity to work correctly. To provide atomic join without relying on transitive connectivity, we divided the join algorithm into three steps. Each of the three steps involves only two peers instead of having a single step requiring the agreement of three peers. This change is crucial because non-transitive connectivity can easily corrupt any protocol involving more than two peers. The division of three steps allow "incomplete" join protocols to be accepted as valid join events introducing branches to the ring. Hence, the name of relaxed ring. These branches appear naturally in presence of connectivity problems in the network, allowing the system to work in realistic scenarios where networks are not perfectly connected. This new topology slightly increases the cost of the routing algorithm, but only in some parts of the ring. Therefore, the global performance is not really affected. These claims are evaluated and validated in Chapter 6. We consider the performance degradation in the routing algorithm a small cost in comparison to the gain in fault tolerance and cost-efficient maintenance.

The relaxed ring topology makes feasible the integration of peers with very poor connectivity. Having a connection to a successor is sufficient to be part of the network. Leaving the network can be done instantaneously without having to follow a *leave* protocol, because the failure-recovery mechanism will deal with the missing node. In Chapter 3, the guarantees and limitations of the system are clearly identified and formally stated providing helpful indications to build fault-tolerant applications on top of this structured overlay network. The relaxed ring is enhanced with a self-adaptable finger table, PALTA, that is able to scale up and down, building a more efficient routing table according to the size of the network.

9.2 Transactional Robust Storage

Our second desired property for scalable systems is transactional robust storage. The basic DHT provided by the relaxed ring has been improved with a replication layer built op top of it. The layer is built using the symmetric replication strategy to place the data items in the network. To guarantee the consistency and coherence of the replicas, a transactional layer called Trappist is in charge of providing atomic updates of the items, with the guarantee that the majority of the replicas store the latest value. Working with the majority of the replicas is a relaxation of classical approaches that requires that all replicas are always updated.

Trappist implements four different transactional protocols, which are described in Chapter 5. They are *two-phase commit*, *Paxos consensus algorithm*, *eager-locking Paxos* and *lock-free key/value-sets*. Two-phase commit is widely used in centralized relational databases, but it does not suit dynamic systems as peer-to-peer networks, because it requires that a single transaction manager survives the transaction, and it also requires that all replicas accept the new value. We have implemented two-phase commit for academic purposes to compare it to our contribution. Paxos-consensus algorithm is taken from the related work, because it works pretty well in peer-to-peer systems thanks to its set of replicated transaction managers, and the fact that it only needs the majority of replicas to commit each transaction. Therefore, it also fits in our relaxed approach. Paxos with eager-locking is a modification of the previous protocol providing a way of eagerly acquiring the lock of the majority of the replicas. Eager locks are useful to build synchronous collaborative applications, where users have highly concurrent access to the shared resources.

We also applied the relaxed approach to improve the support for data collections. We relaxed the order of elements in the collection to provide key/value-sets. Relaxing the versioning of value-sets we were able to provide concurrent modifications without locking the sets, improving performance. We consider that getting rid of distributed locks is an important improvement to deal better with the above mentioned inherent asynchrony of distributed systems. These relaxations were introduced without sacrificing strong consistency and providing transactional support. Evaluation of key/value-sets and the other Trappist's protocols is presented in Chapter 6. The analysis is done in terms of performance and scalability.

9.3 Beernet

The relaxed ring and the Trappist layer are part of the whole implementation of Beernet, a peer-to-peer system where each peer is built as a set of independent components with no shared state that only communicate through message passing. The architecture is achieved by using our own implementation of the actor model. This architecture has the main advantage of preventing errors at the low layers to be propagated to the higher layer. Another advantage is that the architecture can adapt its behaviour dynamically, and therefore, it gives a step forward towards self-configuration of components. In Beernet, every component can be suspended and replaced by a different component with the same interface but with a different behaviour. This change can be done while the rest of components keep on working.

Beernet's design is also modular. For instance, a component implementing a Trappist's protocol does not know anything about the implementation of the DHT. It simply uses its API. It does not know the replication strategy either. Trappist's components delegate the selection of replicas to the replication manager. The fundamental concepts used on the implementation of Beernet and its architecture are described in detail in Chapter 7.

To validate Beernet as programming framework, we have developed two applications taking advantage of all transactional protocols provided in Trappist. We implemented an asynchronous collaborative web service called Sindaca, which is a community driven recommendation system. DeTransDraw is a synchronous tool for collaborative drawing, where each update is done using the eager-locking transactional protocol. Both applications take advantage of the lock-free key/value-sets. A small wiki has been implemented by students using only the API for creating and connecting peers, and for running transactions to manipulate the data of the wiki articles. The management of the peer-to-peer system was transparently provided by Beernet. These three applications are described in Chapter 8.

Given the results presented in this dissertation, and referring to our thesis statement, we can conclude that we have successfully applied the relaxed approach to design and build scalable distributed systems with self-managing behaviour and transactional robust storage.

9.4 Related Work

There are two systems that are strongly related to Beernet: Scalaris [SSR08] and Kompics [AH08]. In this section we briefly described them to state the similarities and differences with out work.

Scalaris This is the most related work that Beernet has. It also provides its own overlay network topology called Chord$^{\#}$, providing multidimensional range queries, and having an infinite circular address space based on lexicographical order of keys. It is the first peer-to-peer system in implementing the Paxos consensus algorithm, using it to build a decentralized clone of the German Wikipedia. That result inspired our student project presented in Section 8.3. The differences with Beernet starts with the network topology. Chord$^{\#}$ uses periodic stabilization for ring maintenance and relies on transitive connectivity to terminate their maintenance algorithms. While they have continued improving the performance of the Paxos consensus algorithm, Beernet has expanded the transactional support with new protocols. More about Scalaris and Chord$^{\#}$ can be read in Section 2.3.4 and 6.6.1.

Kompics The relationship with Kompics lies on the programming model used for building peer-to-peer systems. Kompics also uses independent components that do not share any state, and it builds complex systems starting from basic layers that are aggregated with new components providing new functionality. Kompics is strongly inspired by the work presented in [GR06], and from our point of view, it inherits from Java some limitations in their programming model. For instance, the connection of components using channels with a verbose registration of every single type of event it is going to be transmitted, can be avoided with a simple message passing mechanism between components, as it is done in Beernet.

9.5 Future Work

To conclude this dissertation, we discuss some research ideas derived from the results obtained so far. Each idea explores a different area having different applicabilities, but all three of them have the common denominator of being related to self-management and scalable decentralized systems.

Phase Transitions As we have observed in Section 6.7.3, where we discussed the impact of NAT devices on the skewed distribution of branches, there is a critical point where no peer is allowed to enter the ring anymore. This particular change on state of the network is consistent through the experiments we performed. Therefore, it looks like a phase transition from a dynamic relaxed ring constantly modified by churn, to a more static but unstable relaxed ring that do not allow new peers. We would also like to investigate if there is a clear phase transition triggered by the quality of the network but at a very low rate. If we consider that a relaxed ring in a perfectly connected ring with no churn will form a perfect ring with perfect finger table, it would also be static but much more stable than the other extreme we just discussed. It would be like ice. As soon as churn increases, or the quality of the network decreases, the ring will start moving introducing branches and being less strongly connected, as in liquid. It would be interesting then to find the combination of churn and network quality to be at the point of breaking the ring, when all peers in the successor list are unreachable. This kind of research could lead us to build phase diagrams of peer-to-peer networks as function of churn and network quality.

Cloud Computing Being one of the hottest research topics nowadays, it is difficult not to see the applicability of peer-to-peer systems in Cloud Computing because of the elasticity it provides. Two directions can be followed here. Without changing the current way of doing cloud computing, it is interesting to know how peer-to-peer's ability of scaling up and down can be used to efficiently use the resources hired to Cloud providers. With the concept of *pay-as-you-go* it is not only important to be able to scale up, but also to scale down to released unused resources. Structured overlay networks can do that well. The second direction tries to change the way Cloud Computing is conceived by the large Cloud providers. The goal is to use peer-to-peer systems to connect resources from different organizations and single users, to created a sort of dynamic Cloud which is not managed by any single entity. It breaks the dependency on large Cloud providers, and it reuses available resources that are frequently idle. The idea is not new, but there is still a lot to do in this area.

Self-management at the level of peers The self-organizing, self-healing, and other self-* properties of Beernet are mainly achieved as a global property. The granularity of the changes is always a whole peer. There is not much self-configuration done within each peer. The exceptions are the self-tunable eventually perfect failure detector, and the self-adaptable finger table PALTA. Considering the good modularity of Beernet's architecture, self-configuration appears as a possible research topic to improve the self-management of the system. One idea would be to use Context-Oriented Programming to help the self-adaptable mechanism of the routing table, as in PALTA. In PALTA, there is a switch between two different routing-table strategies determined by the ω value. Below ω, the network works in a context of being fully connected. Above ω, the network moves to a context with logarithmic routing. Therefore, instead of having an *if* statement verifying the value of ω, we could define two contexts to dynamically adapt the behaviour of the finger table whenever the context has changed. Of course, that is not the only way to proceed. Implementing monitoring at the level of components within each peer would help to identify failures in the behaviour of components, allowing the system to replace them, providing self-healing and self-configuration inside each peer, and not only as a global property.

Appendix A

Beernet's API

Beernet is provided as a Mozart-Oz[1] library. To use it, the program needs to import the main functor to create a peer to bootstrap a network, or to join an existing one using the reference of another peer. Importing the main functor and creating a new peer works as follows:

```
functor
import
   Beernet at 'beernet/pbeer/Pbeer.ozf'
define
   Pbeer = {Beernet.new args(transactions:true)}
```

Interacting with the peer is done by triggering an event as follows:

```
{Pbeer event(arg1 ... argn)}
```

We list now the different events that can be triggered on Beernet's peers. Even though Beernet's architecture is organized with layers where Trappist is the upper most one, the architecture does not prevent the access to lower layers because their functionality is important to implement applications.

A.1 Relaxed Ring

The following events can be used to get access to the functionality provided by the relaxed ring layer. It mostly provides access to peer's pointers and other information of the structured overlay network.

A.1.1 Basic Operations

- `join(RingRef)` Triggers joining process using `RingRef` as access point.

[1]The Mozart Programming System, `http://www.mozart-oz.org`

- `lookup(Key)` Triggers lookup for the responsible of `Key`, which will be passed through the hash function.
- `lookupHash(HashKey)` Triggers lookup for the responsible of `HashKey` without passing `HashKey` through the hash function.
- `leave` Roughly quit the network. No gently leave implemented.

A.1.2 Getting Information

- `getId(?Res)`
  Binds `Res` to the id of the peer
- `getRef(?Res)`
  Binds `Res` to a record containing peer's reference with the pattern `pbeer(id:<Id> port:<Port>)`
- `getRingRef(?Res)`
  Binds `Res` to the ring reference
- `getFullRef(?Res)`
  Binds `Res` to a record containing peer's reference and ring's reference with the pattern `ref(pbeer:<Pbeer Ref> ring:<Ring Ref>)`
- `getMaxKey(?Res)`
  Binds `Res` to the maximum key in ring's address space
- `getPred(?Res)`
  Binds `Res` to the reference of peer's predecessor
- `getSucc(?Res)`
  Binds `Res` to the reference of peer's successor
- `getRange(?Res)`
  Binds `Res` to the responsibility range of peer with the pattern `From#To`, where `From` and `To` are integers keys.

A.1.3 Other Events

- `refreshFingers(?Flag)` Triggers lookup for ideal keys of finger table to refresh the routing table. Binds `Flag` when all lookups are replied.
- `injectPermFail` Peer stop answering any message.
- `setLogger(Logger)` Sets Logger as the default service to log information of the peer. Mostly used for testing and debugging.

A.2 Message Sending

This section describe the events that allow applications to send and receive messages to other peers.

- `send(Msg to:Key)`
  Sends message `Msg` to the responsible of key `Key`.
- `dsend(Msg to:PeerRef)`
  Sends a direct message `Msg` to a peer using `PeerRef`.
- `broadcast(Msg range:Range)`
  Sends message `Msg` to all peers on the range `Range`, which can be `all`, sending to the whole ring, `butMe`, sending to all ring except for the sender, and `From#To`, which sends to all peers having an identifier within keys `From` and `To`, so it can be used as a multicast.
- `receive(?Msg)`
  Binds `Msg` to the next message received by the peer, and that it has not been handled by any of Beernet's layer. It blocks until next message is received.

A.3 DHT

Beernet also provides the basic operations of a distributed hash table (DHT). None of this uses replication, therefore, there are no guarantees about persistence.

- `put(Key Val)`
  Stores the value `Val` associated with key `Key`, only in the peer responsible for the key resulting from applying the hash function to key `Key`.
- `get(Key ?Val)`
  Binds `Val` to the value stored with key `Key`. It is bound to the atom **'NOT_FOUND'** in case that no value is associated with such key.
- `delete(Key)`
  Deletes the item associated to key `Key`.

A.4 Symmetric Replication

The symmetric replication layer does not provides an interface to store values with replication, but it does provides some functions to retrieve replicate data, and to send messages to replica-sets.

- `bulk(Msg to:Key)`
  Sends message `Msg` to the replication set associated to key `Key`.

- `getOne(Key ?Val)`
  Binds `Val` to the first answer received from any of the replicas of the item associated with key `Key`. If value is **`'NOT_FOUND'`**, the peer does not bind `Val` until it gets a valid value, or until all replicas has replied **`'NOT_FOUND'`**.

- `getAll(Key ?Val)`
  Binds `Val` to a list containing all values stored in the replica set associated to key `Key`.

- `getMajority(Key ?Val)`
  Binds `Val` to a list containing the values from the replica set associated to key `Key`. It binds `Val` as soon as the majority is reached.

A.5 Trappist

Trappist provides different protocols to run transactions on replicated items. Due to their specific behaviour, they have different interfaces.

A.5.1 Paxos Consensus

- `runTransaction(Trans Client Protocol)`
  Run the transaction `Trans` using protocol `Protocol`. The answer, `commit` or `abort` is sent to the port `Client`. Currently, the protocols supported by this interface are `twophase` and `paxos`, for two-phase commit and Paxos consensus with optimistic locking. For eager locking, see the interface in Section A.5.2.

- `executeTransaction(Trans Client Protocol)`
  Exactly the same as `runTransaction`. Kept only for backward compatibility.

Inside a transaction, there are three operations that can be used to manipulate data.

- `write(Key Val)`
  Write value `Val` using key `Key`. The new value is stored at least in the majority of the replicas. Updating the value gives a new version number to the item.

- `read(Key ?Val)`
  Binds `Val` to the latest value associated to key `Key`. Strong consistency is guaranteed by reading from the majority of the replicas.

- `remove(Key)`
  Removes the item associated to key `Key` from the majority of the replicas.

A.5.2 Paxos with Eager Locking

- `getLocks(Keys ?LockId)`
  Get the locks of the majority of replicas of all items associated to the list of keys `Keys`. Binds `LockId` to an Oz name if locks are successfully granted, and to `error` otherwise.

- `commitTransaction(LockId KeyValuePairs)` update all items of the list `KeyValuePairs` which must be locked using `LockId`. Each element of the list `KeyValuePairs` must be of the form `<key>#<value>`

A.5.3 Notification Layer

- `becomeReader(Key)`
  Subscribes the current peer to be notified about locking and updates of the item associated to key `Key`. Notification are received using the `receive` event described previously.

A.5.4 Key/Value-Sets

- `add(Key Val)`
  Adds the value `Val` to the set associated to key `Key`.

- `remove(Key Val)`
  Removes value `Val` from the set associated to key `Key`.

- `readSet(Key ?Val)`
  Binds `Val` to a list containing all elements from the set associated to key `Key`.

Appendix B

Sindaca User Guide

Sindaca is a community driven recommendation system for Sharing Idols N Discussing About Common Addictions. In this application, users can make recommendations on music, videos, text and other cultural expressions. It is not designed for file sharing to avoid legal issues with copyright. It allows users to provide links to official sources of titles. Users get notifications about new suggestions, and they can vote on the suggestions to express their preferences. Sindaca is available for testing at url:

`http://beernet.info.ucl.ac.be/sindaca`

Check the contact information of the page to request a username to login.

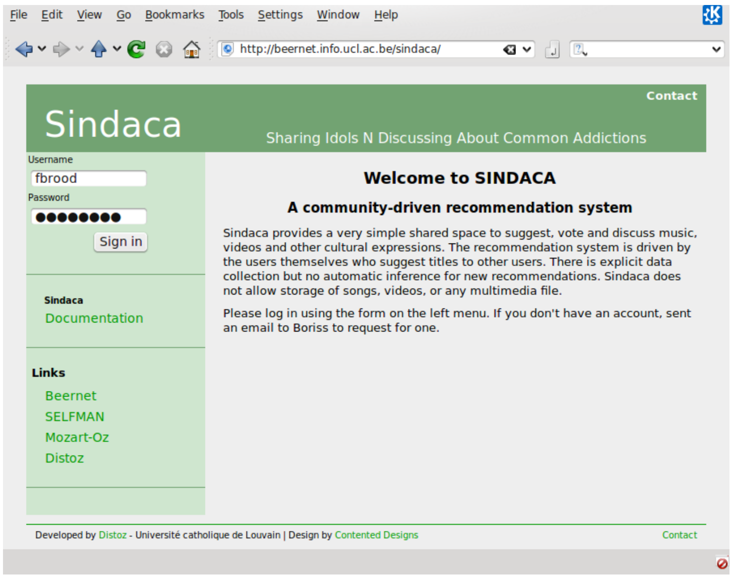
Any modifications done by the tester user will be stored in the network, but they are not persistent to the reinitialization of the network. In case of problems during the test, please check contact information on the web page.

B.1 Signing In

Figure B.1 shows Sindaca's welcome page with the sign-in form on the left of the page, together with the menu. The screenshot shows user *fbrood* logging in.

B.2 After Sign-In and Voting

If username and password are successfully provided, the user is taken to the profile page where information about the recommendations stored in the system is displayed. Figure B.2 is a screenshot of the web page displayed after user *fbrood* has signed in. There is a welcome message both on the menu and on the center of the content. What follows is a list of recommendations suggested by members of the Sindaca community. This recommendation could have been


File Edit View Go Bookmarks Tools Settings Window Help
http://beernet.info.ucl.ac.be/sindaca/
Contact
Sindaca
Sharing Idols N Discussing About Common Addictions
Username
fbrood
Password
Sign in
Sindaca
Documentation
Links
Beernet
SELFMAN
Mozart-Oz
Distoz

Welcome to SINDACA

A community-driven recommendation system

Sindaca provides a very simple shared space to suggest, vote and discuss music, videos and other cultural expressions. The recommendation system is driven by the users themselves who suggest titles to other users. There is explicit data collection but no automatic inference for new recommendations. Sindaca does not allow storage of songs, videos, or any multimedia file.

Please log in using the form on the left menu. If you don't have an account, sent an email to Boriss to request for one.

Developed by Distoz - Université catholique de Louvain | Design by Contented Designs Contact

Figure B.1: Sindaca's welcome page with sign in form.

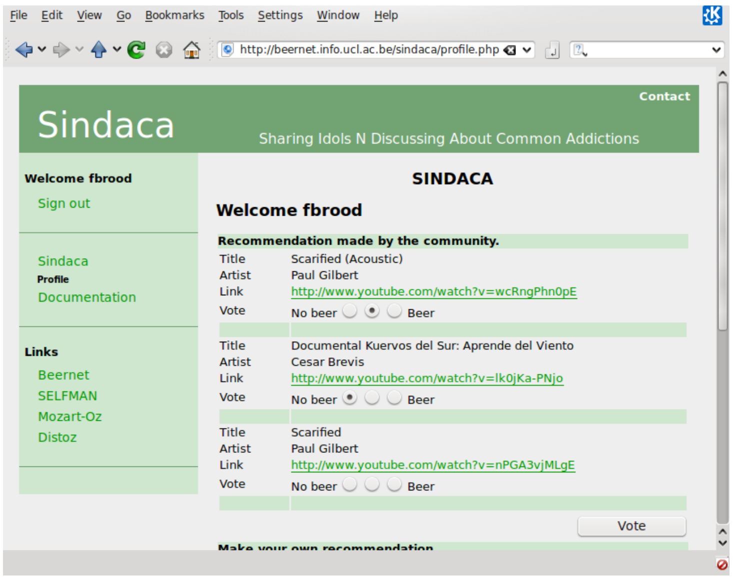
made by other members or by the user itself. The recommendation is composed of a title, the name of the artist, and a link where the title can be found. As mentioned before, Sindaca does not provide storage for content preventing legal issues with copyright.

The listed recommendations are only those that has not received a vote from the user. A radiobutton is provided to express the preference which goes from no good to good. We have actually chosen a scale from *No beer* to *Beer*. The votes are submitted to the network when the user press the `Vote` button. Once the voting submission is sent, a transaction is triggered to modify the item that stores the recommendation.

B.3 Making a Recommendation

The form to add a new recommendation is presented in the same page where the recommendations to be voted are displayed. The form can be seen in Figure B.3 where the data for a new recommendation is already completed. The user must fill in the title, author, and link to the title. Once the data is submitted by clicking the button `Recommend`, a new item will be created in the network storing the recommendation. This item will be associated to the user that creates it.

Every user has a list of recommendations she has made. This list is displayed in the same profile page, below the form for adding new recommendations. Therefore, the full profile page displays from top to bottom: welcome message,



http://beernet.info.ucl.ac.be/sindaca/profile.php

Contact

Sindaca

Sharing Idols N Discussing About Common Addictions

Welcome fbrood

Sign out

Sindaca

Profile

Documentation

Links

Beernet

SELFMAN

Mozart-Oz

Distoz

SINDACA

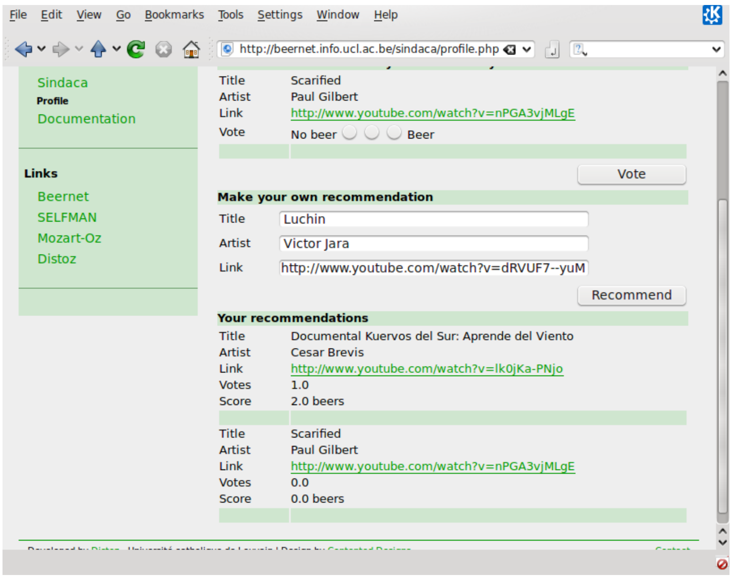
Welcome fbrood

Recommendation made by the community.

Title	Scarified (Acoustic)
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=wcRngPhn0pE
Vote	No beer ○ ● ○ Beer
Title	Documental Kuervos del Sur: Aprende del Viento
Artist	Cesar Brevis
Link	http://www.youtube.com/watch?v=lk0jKa-PNjo
Vote	No beer ● ○ ○ Beer
Title	Scarified
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=nPGA3vjMLgE
Vote	No beer ○ ○ ○ Beer

Vote

Figure B.2: After sign in, users can vote for suggested recommendations.



http://beernet.info.ucl.ac.be/sindaca/profile.php

Sindaca

Profile

Documentation

Links

Beernet

SELFMAN

Mozart-Oz

Distoz

Title	Scarified
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=nPGA3vjMLgE
Vote	No beer ○ ○ ○ Beer

Vote

Make your own recommendation

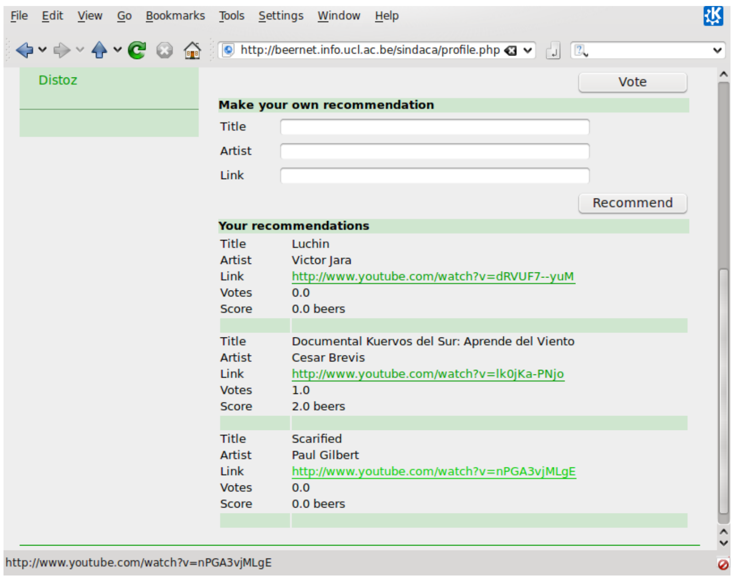
Title	Luchin
Artist	Victor Jara
Link	http://www.youtube.com/watch?v=dRVUF7--yuM

Recommend

Your recommendations

Title	Documental Kuervos del Sur: Aprende del Viento
Artist	Cesar Brevis
Link	http://www.youtube.com/watch?v=lk0jKa-PNjo
Votes	1.0
Score	2.0 beers
Title	Scarified
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=nPGA3vjMLgE
Votes	0.0
Score	0.0 beers

Figure B.3: Adding a new recommendation.


File Edit View Go Bookmarks Tools Settings Window Help
http://beernet.info.ucl.ac.be/sindaca/profile.php
Distoz
Vote
Recommend

Make your own recommendation

Title	
Artist	
Link	

Your recommendations

Title	Luchin
Artist	Victor Jara
Link	http://www.youtube.com/watch?v=dRVUF7--yuM
Votes	0.0
Score	0.0 beers
Title	Documental Kuervos del Sur: Aprende del Viento
Artist	Cesar Brevis
Link	http://www.youtube.com/watch?v=lk0jKa-PNjo
Votes	1.0
Score	2.0 beers
Title	Scarified
Artist	Paul Gilbert
Link	http://www.youtube.com/watch?v=nPGA3vjMLgE
Votes	0.0
Score	0.0 beers

http://www.youtube.com/watch?v=nPGA3vjMLgE

Figure B.4: State of recommendations proposed by the user.

list of recommendations to be voted, form to add a new recommendation, and the list of recommendations already made by the user. Figure B.4 shows how the last list is presented. Apart from the above mentioned fields, namely title, artist and link, the information contains two other fields being part of the state of every recommendation: the amount of votes, and the average score of the title. The screenshot we display in Figure B.4 was taken after the addition of the recommendation made in Figure B.3. For that item we can observe that no vote is registered, and therefore there is no average score either.

B.4 Closing Words

Sindaca has been a successful proof-of-concept for Beernet and its transactional layer but it is still under development, so it is not complete stable, and there are several bugs to be fixed. There are also many new features that can be developed, and we are also planning to have a Mozart-Oz client that can take advantage of the eager notification feature.

Bibliography

[AAG$^+$05] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Seif Haridi, and Manfred Hauswirth. The essence of p2p: A reference architecture for overlay networks. In Germano Caronni, Nathalie Weiler, Marcel Waldvogel, and Nahid Shahmehri, editors, *Peer-to-Peer Computing*, pages 11–20. IEEE Computer Society, 2005.

[AB95] MySQL AB. MySQL: The world's most popular open source database. `http://www.mysql.com`, 1995.

[AEABH03] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 344, Washington, DC, USA, 2003. IEEE Computer Society.

[AFG$^+$09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Feb 2009.

[AH02] Karl Aberer and Manfred Hauswirth. An overview on peer-to-peer information systems. In *WDAS-2002 Proceedings*. Carleton Scientific, 2002.

[AH08] Cosmin Arad and Seif Haridi. Practical protocol composition, encapsulation and sharing in kompics. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:266–271, 2008.

[Ama09] Amazon. Amazon web services. `http://aws.amazon.com`, 2009.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[Arm96] J. Armstrong. Erlang –- a survey of the language and its industrial applications. In *INAP'96 –- The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.

[Aud01] Audiogalaxy. The audiogalaxy satellite. `http://www.audiogalaxy.com/satellite/`, 2001.

[Bar03] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume, reissue edition, April 2003.

[BCvR09] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[BL03] Chonggang Wang Bo and Bo Li. Peer-to-peer overlay networks: A survey. Technical report, 2003.

[Bul09] Alexandre Bultot. A survey of systems with multiple interacting feedback loops and their application to programming. Master's thesis, École Polytechnique de Louvain, Université catholique de Louvain, 2009.

[Cho04] Chord Developers. The Chord/DHash project. `http://pdos.csail.mit.edu/chord/`, 2004.

[CM04] Bruno Carton and Valentin Mesaros. Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par*, volume 3149, pages 1060–1067. Springer, 2004.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving DNS using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[CMV$^+$08] Alfredo Cádiz, Boris Mejías, Jorge Vallejos, Kim Mens, Peter Van Roy, and Wolfgang De Meuter. PALTA: Peer-to-peer AdaptabLe Topology for Ambient intelligence. In M. Cecilia Bastarrica and Mauricio Solar, editors, *SCCC*, pages 100–109. IEEE Computer Society, 2008.

[Col07] Raphaël Collet. *The Limits of Network Transparency in a Distributed Programming Language.* PhD thesis, Université catholique de Louvain, dec 2007.

[CV06] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[DBK+01] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David R. Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *HotOS*, pages 81–86. IEEE Computer Society, 2001.

[DGM02] Neil Daswani and Hector Garcia-Molina. Query-flood dos attacks in gnutella. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 181–192, New York, NY, USA, 2002. ACM.

[Dis09] Distributed Systems Architecture Research Group at Universidad Complutense de Madrid. Opennebula. `http://www.opennebula.org`, 2009.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[DL09] Cameron Dale and Jiangchuan Liu. apt-p2p: A peer-to-peer distribution system for software package releases and updates. In *IEEE INFOCOM*, Rio de Janeiro, Brazil, April 2009.

[Dou02] John R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[DPS09] L. D'Acunto, J. A. Pouwelse, and H. J. Sips. A measurement of nat and firewall characteristics in peer-to-peer systems. In Lex Wolters Theo Gevers, Herbert Bos, editor, *Proc. 15-th ASCI Conference*, pages 1–5, P.O. Box 5031, 2600 GA Delft, The Netherlands, June 2009. Advanced School for Computing and Imaging (ASCI).

[DVM+06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'hondt, and Wolfgang De Meuter. *Ambient-Oriented Programming in AmbientTalk.* 2006.

[EAH05] Sameh El-Ansary and Seif Haridi. An overview of structured overlay networks. In *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks.* 2005.

[Emu04] Emule. The emule file-sharing application homepage. `http://www.emule-project.org`, 2004.

[FFME04] Michael Freedman, Eric Freudenthal, David Mazières, and David Mazi Eres. Democratizing content publication with coral. In *In NSDI*, pages 239–252, 2004.

[FI03] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03*, pages 118–128, 2003.

[FLRS05] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and dhts. In *WORLDS'05: Proceedings of the 2nd conference on Real, Large Distributed Systems*, pages 55–60, Berkeley, CA, USA, 2005. USENIX Association.

[Fre03] FreeNet Community. The freenet project. `http://freenetproject.org`, 2003.

[Gat97] Erann Gat. On three-layer architectures. In *ARTIFICIAL INTELLIGENCE AND MOBILE ROBOTS*, pages 195–210. AAAI Press, 1997.

[GDA06] Sarunas Girdzijauskas, Anwitaman Datta, and Karl Aberer. Oscar: Small-world overlay for realistic key distributions. In Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and Aris M. Ouksel, editors, *DBISP2P*, volume 4125 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2006.

[GGG$^+$03] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, New York, NY, USA, 2003. ACM.

[GGV05] Donatien Grolaux, Kevin Glynn, and Peter Van Roy. Lecture notes in computer science. In Peter Van Roy, editor, *MOZ*, volume 3389, pages 149–160. Springer, 2005.

[Gho06] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD thesis, KTH — Royal Institute of Technology, Stockholm, Sweden, dec 2006.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, page 2002, 2002.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[Gly05] Kevin Glynn. Extending the oz language for peer-to-peer computing. Technical report, Université catholique de Louvain, Belgium, March 2005.

[Gly07] Kevin Glynn. P2pkit: A services based architecture for deploying robust peer-to-peer applications. `http://p2pkit.info.ucl.ac.be/index.html`, 2007.

[GMV07] Donatien Grolaux, Boris Mejías, and Peter Van Roy. PEPINO: PEer-to-Peer network INspectOr. In Hauswirth et al. [HWW$^+$07], pages 247–248.

[Gnu03] Gnutella. Gnutella. `http://www.gnutella.com`, 2003.

[GOH04] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. A novel replication scheme for load-balancing and increased security. Technical Report T2004:11, Swedish Institute of Computer Science (SICS), June 2004.

[Goo09] Google Inc. Google app engine. `http://code.google.com/appengine/`, 2009.

[GR06] Rachid Guerraoui and Louis Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

[Gro98] Donatien Grolaux. Editeur graphique réparti basé sur un modéle transactionnel, 1998. Mémoire de Licence.

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, New York, NY, USA, 2002. ACM.

[HFC$^+$08] Yan Huang, Tom Z. J. Fu, Dah-Ming Chiu, John C. S. Lui, and Cheng Huang. Challenges, design and analysis of a large-scale p2p-vod system. *SIGCOMM Comput. Commun. Rev.*, 38(4):375–388, 2008.

[HWW$^+$07] Manfred Hauswirth, Adam Wierzbicki, Klaus Wehrle, Alberto Montresor, and Nahid Shahmehri, editors. *Seventh IEEE International Conference on Peer-to-Peer Computing (P2P 2007), September 2-5, 2007, Galway, Ireland*. IEEE Computer Society, 2007.

[HWY08] Felix Halim, Yongzheng Wu, and Roland H. C. Yap. Security issues in small world network routing. In *SASO '08: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive* *and Self-Organizing Systems*, pages 493–494, Washington, DC, USA, 2008. IEEE Computer Society.

[IF04] Adriana Iamnitchi and Ian Foster. A peer-to-peer approach to resource location in grid environments. pages 413–429, 2004.

[JOK09] R. Jimenez, F. Osmani, and B. Knutsson. Connectivity properties of mainline bittorrent dht nodes. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 262–270, 2009.

[KA08] Supriya Krishnamurthy and John Ardelius. An analytical framework for the performance evaluation of proximity-aware structured overlays. Technical report, Swedish Institute of Computer Science (SICS), Sweden, 2008.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[KMV06] Erik Klintskog, Boris Mejías, and Peter Van Roy. Efficient distributed objects by freedom of choice. In *Revival of Dynamic Languages Workshop, ECOOP'06*, July 2006.

[KPQS09] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. Nat-resilient gossip peer sampling. *Distributed Computing Systems, International Conference on*, 0:360–367, 2009.

[LCP+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[LML05] Paul Lin, Alexander MacArthur, and John Leaney. Defining autonomic computing: A software engineering perspective. pages 88–97, Brisbane, Australia, March 31 - April 1 2005. IEEE Computer Society.

[LMP04] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.

[LMP06] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2):126–148, 2006.

[LP09] Yangyang Liu and Jianping Pan. The impact of nat on bittorrent-like p2p systems. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 242–251, 2009.

[LQK$^+$08] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang. Inside the new coolstreaming: Principles, measurements and performance implications. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008.

[Mar02] Evangelos P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[Mat04] MathWorld. The butterfly graph. `http://mathworld.wolfram.com`, 2004.

[MCGV05] Valentin Mesaros, Raphael Collet, Kevin Glynn, and Peter Van Roy. A transactional system for structured overlay networks. Technical report, March 2005.

[MCPV05] Boris Mejías, Raphaël Collet, Konstantin Popov, and Peter Van Roy. Improving transparency of a distributed programming system. In *"Integrated Research in Grid Computing" CoreGRID Workshop*, November 2005.

[MCV05] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for mozart. In Peter Van Roy, editor, *MOZ*, volume 3389, pages 125–136. Springer, 2005.

[MCV09] Boris Mejías, Alfredo Cádiz, and Peter Van Roy. Beernet: RMI-free peer-to-peer networks. In *DO21 '09: Proceedings of the 1st International Workshop on Distributed Objects for the 21st Century*, pages 1–8, New York, NY, USA, 2009. ACM.

[Mej09] Boris Mejías. Beernet: a relaxed-ring approach for peer-to-peer networks with transactional replicated DHT. Doctoral Symposium at the XtreemOS Summer School 2009, Wadham College, University of Oxford, Oxford, UK, September 2009.

[MGV07] Boris Mejías, Donatien Grolaux, and Peter Van Roy. Improving the peer-to-peer ring for building fault-tolerant grids. In *CoreGRID Workshop on Grid-* and P2P-**, july 2007.

[MH07] Monika Moser and Seif Haridi. Atomic Commitment in Transactional DHTs. In *Proceedings of the CoreGRID Symposium. CoreGRID series*. Springer, 2007.

[MHV08] Boris Mejías, Mikael Högqvist, and Peter Van Roy. Visualizing transactional algorithms for DHTs. In Klaus Wehrle, Wolfgang Kellerer, Sandeep K. Singhal, and Ralf Steinmetz, editors, *The Eighth IEEE International Conference on Peer-to-Peer Computing*, pages 79–80. IEEE Computer Society, 2008.

[Mic09] Microsoft Corporation. Azure service platform. `http://www.microsoft.com/azure/`, 2009.

[Mil05] Drazen Milicic. *Software quality models and philosophies*, chapter 1, page 100. Blekinge Institute of Technology, June 2005.

[MJV06] Boris Mejías, Yves Jaradin, and Peter Van Roy. Improving robustness in P2PS and a generic belief propagation service for P2PKit. Technical report, Department of Computing Science and Engineering, Université catholique de Louvain, December 2006.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. pages 31–44, 2002.

[MMR95] Martin Müller, Tobias Müller, and Peter Van Roy. Multiparadigm programming in oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 dec 1995. A Workshop in Association with ILPS'95.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. pages 183–192, 2002.

[Moz08] Mozart Consortium. The mozart-oz programming system. `http://www.mozart-oz.org`, 2008.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In Vijay Saraswat, editor, *ASIAN'03*. Springer Verlag, December 2003.

[MTSL05] Mark S. Miller, E. Dean Tribble, Jonathan Shapiro, and Hewlett Packard Laboratories. Concurrency among strangers: Programming in e as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[MV07] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *SCCC '07: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[MV08] Boris Mejías and Peter Van Roy. The relaxed-ring: a fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, 2008.

[MV10] Boris Mejías and Peter Van Roy. Beernet: Building self-managing decentralized systems with replicated transactional storage. *IJARAS: International Journal of Adaptive, Resilient, and Autonomic Systems*, 1(3):1–24, July - September 2010. to appear.

[Nap99] Napster, Inc. Napster. `http://www.napster.com`, 1999.

[Ope01] OpenNap Community. Open source napster server. `http://opennap.sourceforge.net`, 2001.

[Ove04] Overnet. The overnet file-sharing application homepage. `http://www.overnet.com`, 2004.

[Par09] XtreemOS Partners. XtreemOS: Enabling Linux for the grid. `http://www.xtreemos.org`, 2009.

[PGES05] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. 2005.

[Pos09] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. `http://www.postgresql.org/`, 2009.

[Pro08] Programming Languages and Distributed Computing Research Group, UCLouvain. P2ps: A peer-to-peer networking library for mozart-oz. `http://gforge.info.ucl.ac.be/projects/p2ps/`, 2008.

[Pro09] Programming Languages and Distributed Computing Research Group, UCLouvain. Beernet: pbeer-to-pbeer network. `http://beernet.info.ucl.ac.be`, 2009.

[PRS07] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. In *Managing Virtualization of Networks and Services*, pages 256–267. 2007.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, 2001.

[REAH09] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Natcracker: Nat combinations matter. *Computer Communications and Networks, International Conference on*, 0:1–7, 2009.

[Res09] Reservoir Consortium. Reservoir: Resources and services virtualization without barriers. `http://www.reservoir-fp7.eu`, 2009.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[RFI02] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system. *IEEE Internet Computing Journal*, 6:2002, 2002.

[RGK$^+$05] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: A public dht service and its uses, 2005.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *In Proceedings of the USENIX Annual Technical Conference*, 2004.

[SAZ$^+$02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure, 2002.

[SGH07] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Handling network partitions and mergers in structured overlay networks. In Hauswirth et al. [HWW$^+$07], pages 132–139.

[SGH08] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with network partitions in structured overlay networks. *Journal of Peer-to-Peer Networking and Applications*, 2008.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[Smo95] Gert Smolka. The oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, chapter Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[SMS$^+$08] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-based consistency and availability in structured overlay networks. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08)*. ACM, jun 2008.

[SRHS10] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schutt. Enhanced paxos commit for transactions on dhts. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:448–454, 2010.

[SSR07] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. A structured overlay for multi-dimensional range queries. In *Euro-Par 2007*, 2007.

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[ST05] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing: emerging trends and open problems. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[The03] The PlanetLab Consortium. Planetlab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`, 2003.

[The09a] The Apache Software Foundation. Apache http server. `http://www.apache.org`, 2009.

[The09b] The Globus Alliance. Nimbus Open Source IaaS Cloud Computing Software. `http://workspace.globus.org/`, 2009.

[The09c] The PHP Group. PHP: Hypertext Preprocessor. `http://www.php.net`, 2009.

[TT03] Domenico Talia and Paolo Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(4):96–94, 2003.

[TTF$^+$06] Paolo Trunfio, Domenico Talia, Paraskevi Fragopoulou, Charis Papadakis, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Seif Haridi. Peer-to-peer models for resource discovery on grids. In *Proc. of the 2nd CoreGRID*

Workshop on Grid and Peer to Peer Systems Architecture, Paris, France, January 2006.

[TTH+07] Paolo Trunfio, Domenico Talia, Seif Haridi, Paraskevi Fragopoulou, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Harris Papadakis. Peer-to-peer resource discovery in grids: Models and systems. *Future Generation Computer Systems*, 23(7):864–878, August 2007.

[TTZH06a] Domenico Talia, Paolo Trunfio, Jingdi Zeng, and Mikael Högqvist. A dht-based peer-to-peer framework for resource discovery in grids. Technical Report TR-0048, June 2006.

[TTZH06b] Domenico Talia, Paolo Trunfio, Jingdi Zeng, and Mikael Högqvist. A peer-to-peer framework for resource discovery in large-scale grids. In *Proc. of the 2nd CoreGRID Integration Workshop*, pages 249–260, Krakow, Poland, October 2006.

[TV01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Van06] Peter Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, September 2006.

[Van08] Peter Van Roy. Overcoming software fragility with interacting feedback loops and reversible phase transitions. 2008.

[Van09] Peter Van Roy. Languages and algorithms for distributed applications. `http://www.info.ucl.ac.be/Enseignement/Cours/SINF2345/`, 2009.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[VMG+07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. *Chilean Computer Science Society, International Conference of the*, 0:3–12, 2007.

[WER91] Daniel M. Wegner, Ralph Erber, and Paula Raymond. Transactive memory in close relationships. *Journal of Personality and Social Psychology*, 61(6):923–929, December 1991.

[Wik09] Wikimedia Foundation. Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Wikipedia`, 2009.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1994.

[ZHS$^+$03] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.