UNIVERSITÉ DE NEUCHÂTEL

SPLAY: A toolkit for the design and evaluation of Large Scale Distributed Systems

Thèse

présentée le 27 juin 2014 à la Faculté des Sciences de l'Université de Neuchâtel pour l'obtention du grade de **docteur ès sciences** par

Lorenzo Leonini

acceptée sur proposition du jury:

Prof. Pascal Felber	Université de Neuchâtel	directeur de thèse
Prof. Peter Kropf	Université de Neuchâtel	rapporteur
Dr Etienne Rivière	Université de Neuchâtel	rapporteur
Prof. Vivien Quéma	INP, Grenoble	rapporteur
Prof. Spyros Voulgaris	VU University, Amsterdam	rapporteur

UNIVERSITÉ DE NEUCHÂTEL

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

IMPRIMATUR POUR THESE DE DOCTORAT

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

Monsieur Lorenzo LEONINI

Titre:

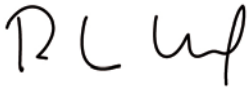
“Splay: A toolkit for the design and evaluation of Large Scale Distributed Systems”

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, Université de Neuchâtel, directeur de thèse
- Prof. Peter Kropf, Université de Neuchâtel
- Dr Etienne Rivière, Université de Neuchâtel
- Prof. Vivien Quéma, INP, Grenoble, France
- Prof. Spyros Voulgaris, VU University, Amsterdam, Pays-Bas

Neuchâtel, le 30 avril 2014

Le Doyen, Prof. P. Kropf



To my family.

Acknowledgements

First, I would like to thank my adviser, Prof. Pascal Felber, for all his guiding, advice, understanding and for our many shared sporting activities during my PhD studies.

I would also like to thank Dr. Etienne Rivière for our collaboration and all the nice moments spent together.

A special thank to my delicious office mate, Sabina, for her humor, patience and great discussions.

Another special thank to Prof. Peter Kropf for all his ~~threats~~ encouragement.

Finally, I want to thank all my professors, colleagues and friends at the Computer Science Department for the great moments at the university and outdoors.

Abstract

Keywords: Distributed Systems, Distributed Algorithms, Peer-to-peer network, LSDS, Large Scale Experiments, Deployment, Sandbox, Simulation, Gossip-based dissemination, Epidemic algorithm, Relevance feedback

This thesis presents SPLAY, an integrated system that facilitates the design, deployment and testing of large-scale distributed applications.

SPLAY covers all aspects of the development and evaluation chain.

It allows developers to express algorithms in a concise, simple language that highly resembles pseudo-code found in research papers. The execution environment has low overheads and footprint, and provides a comprehensive set of libraries for common distributed systems operations.

SPLAY applications are run by a set of daemons distributed on one or several testbeds. They execute in a sandboxed environment that shields the host system and enables SPLAY to also be used on non-dedicated platforms, in addition to classical testbeds like PlanetLab or ModelNet.

We illustrate the interest of SPLAY for distributed systems research by covering two representative examples. First, we present the design and evaluation of PULP, an efficient generic push-pull dissemination protocol which combines the best of pull-based and push-based approaches. PULP exploits the efficiency of push approaches, while limiting redundant messages and therefore imposing a low overhead, as pull protocols do. PULP leverages the dissemination of multiple messages from diverse sources: by exploiting the push phase of messages to transmit information about other disseminations, PULP enables an efficient pulling of other messages, which themselves help in turn with the dissemination of pending messages.

Finally, we present the design and evaluation of a collaborative search companion system, CoFeed, that collects user search queries and accesses feedback to build user and document-centric profiling information. Over time, the system constructs ranked collections of elements that maintain the required information diversity and enhance the user search experience by presenting additional results tailored to the user interest space.

Résumé

Mots-clés: Systèmes distribués à grande échelle, Algorithmes distribués, Réseaux pair à pair, Expérimentations à grande échelle, Déploiement, Bac à sable, Simulation, Algorithmes épidémiques

Cette thèse présente SPLAY, un système intégré qui facilite la conception, le déploiement et les expérimentations des systèmes distribués à grande échelle. SPLAY couvre toutes les étapes du développement à l'évaluation.

Il permet à des développeurs d'exprimer des algorithmes de manière simple et concise dans un langage proche du pseudo-code que l'on peut trouver dans les publications scientifiques. L'environnement d'exécution est léger et fournit un ensemble de librairies répondant aux principaux besoins pour la conception de systèmes distribués.

Les applications SPLAY sont exécutées par un ensemble de processus distribués sur un ou plusieurs systèmes de test. Ils exécutent ensuite l'application au sein d'un environnement confiné, ce qui permet d'utiliser SPLAY sans risques même sur des plates-formes non dédiées en plus des environnements classiques tels que PlanetLab ou ModelNet.

Nous illustrons l'intérêt de SPLAY pour la recherche sur les systèmes distribués à l'aide de deux exemples représentatifs.

Tout d'abord, nous décrivons la conception et l'évaluation de PULP, un protocole de dissémination efficace qui combine le meilleur des approches "pousser" et "tirer". PULP exploite l'efficacité de l'approche "pousser" tout en en limitant la redondance par l'usage de l'approche "tirer" dont la fréquence est conditionnée par des informations complémentaires jointes aux paquets de données.

Finalement, nous présentons la conception et l'évaluation d'un système d'aide à la recherche, CoFeed, qui collecte les recherche des utilisateurs et les accès effectués afin de construire un profil d'utilisateur et de documents. Au fil du temps, le système crée des collections triées de documents qui permettent d'améliorer la qualité des recherches en fournissant des résultats complémentaires correspondant aux domaines d'intérêt de l'utilisateur.

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

There seems no plan because it is all plan.

C. S. Lewis

1.1 Context

A distributed system refers to a set of computing entities collaborating for a task by communicating through a network. A client-server system is the simplest of distributed systems, but we consider in this thesis the more general case of a set of components that together create a virtual networked architecture, sharing various of their resources, in order to reach a common goal, in a collaborative manner.

Recently, we observe an unprecedented growth in the number and the scale and importance of deployed distributed systems. This is in great part due to the extreme increase in the amount of data that need to be stored and processed, a phenomenon which is often refereed to as *Big Data*. Traditional architectures such as client-server and SQL databases cannot handle such huge workloads appropriately.

Distributed systems provide effective solutions to face these new challenges in particular because of their support for scalability.

1. INTRODUCTION

One can broadly characterize distributed systems as either loosely or tightly coupled systems. P2P (peer-to-peer) systems lie in the first category. In these systems, each user or host shares some resources to become a part and have access to the system. Here, the system could be designed to scale with the number of its users. The second category includes enterprise systems like clusters and server farms. Scalability is achieved by having the ability to transparently add resources to the existing set.

Distributed architectures rarely provide the same functionalities and guarantees that were previously available with centralized systems (the main reason being that providing such guarantees would ruin scalability and speed). To benefit from a distributed system, all the software stack has to be rethought.

It is very interesting to observe that, in parallel to the distributed systems challenge, the software industry also faces another challenge that shares some similarities: modern software has to scale with the number of processor cores available inside a single computer especially the concurrency problems. This seems to indicate the end of vertical scalability; which was obtained by replacing components with more powerful ones. Instead, horizontal scalability is achieved by adding more components that will cooperate with the existing ones.

For several years, many papers have been published describing distributed algorithms for large-scale systems, for example, DHT (Distributed Hash Tables) in P2P networks [105, 121, 110, 134] . These papers often describe clever algorithms provided to the reader in pseudo-code format. During our studies of distributed systems, we encountered on many occasions a huge problem related to these papers: the difficulty to validate their algorithms and reproduce performance results.

The gap between a theoretical pseudo-code and a working/testable implementation is very important. Many problems are encountered during this process: writing a working implementation (many real life problems, notably network delays and failures, are not in the pseudo-code), testing it on a simulator, deploying it on various testbeds, analyzing the results, and repeating this process until results.

Various languages, tools and testbeds help in that process. However, these components are far from providing a unified environment. Their specializations often limit them to a specific category of experiments. It is also difficult to combine them or to reuse some code when switching from one to another.

1.2 Contributions

To address these limitations, we propose SPLAY, an infrastructure that simplifies the prototyping, development, deployment and evaluation of large-scale systems. Unlike existing tools, SPLAY covers the whole chain of distributed systems design and evaluation. It allows developers to specify distributed applications concisely using a platform-independent, lightweight and efficient language based on Lua [60].

SPLAY provides a secure and safe environment for executing and monitoring applications, and allows for a simplified and unified usage of testbeds such as PlanetLab [1], ModelNet [127], networks of idle workstations, or personal computers.

SPLAY applications execute in a safe, sandboxed environment with controlled access to local resources (file system, network, memory) and can be instantiated on a large set of nodes with a single command. SPLAY supports multi-user resource reservation and selection, orchestrates the deployment and monitors the whole system. It is particularly easy with SPLAY to reproduce a given live experiment or to control several experiments at the same time.

An important component of SPLAY is its churn manager, which can reproduce the dynamics of a distributed system based on real traces or synthetic descriptions. This aspect is of primordial importance, as natural churn present in some testbeds such as PlanetLab is not reproducible, hence preventing a fair comparison of protocols under the very same conditions.

The system has been thoroughly evaluated along all its aspects: concision and facility of development, efficiency, scalability and features. Experiments convey SPLAY's good properties and the ability of the system to help the practitioner or researcher through the whole distributed system design, implementation and evaluation chain, among which lie the following examples:

1. Testing overlays and other large-scale distributed systems, whose lifetime is specified at runtime and usually short. In such a situation, one needs to get the results (i.e., logs) as fast and as easily as possible, and the deployment needs to be quick and reliable. A typical example is the analysis of the diffusion of a file using the BitTorrent protocol [42], where the experiment can be terminated once all nodes have finished downloading the file.

2. Running long-term applications, such as indexing services or DHTs. For instance, one may want to run in the long-term a DHT such as Pastry [110], this service being used by another application under test. A potential difficulty can stem

in the need to keep a fixed size set of nodes participating in the DHT. SPLAY facilities for node management make such a task easy.

3. Deploying short-term applications on a network of workstations as part of some course's lab work. In such a situation, resource usage at the nodes must be controlled to avoid any impact on the workstations' systems. Such a task would require much administration, especially with heterogeneous operating systems at testbeds' nodes. SPLAY sandboxed execution environment in turn permits us to consider a broader range of testbeds for distributed systems evaluation.

The file transfer scenario is an emblematic example of our original intent when designing SPLAY: an overlay must be deployed for a task that is both occasional and relatively short. Without appropriate tools, the effort of constructing such an overlay is disproportionate compared to the task.

Applications are concise (e.g., less than 100 lines of code for a complete implementation of Chord [121]), platform-independent, lightweight and efficient. SPLAY applications have access to a comprehensive set of libraries tailored for the development of distributed protocols and overlays.

SPLAY has also been used as the development and evaluation framework for several research effort at the university of Neuchâtel and other universities worldwide. We provide two representative examples of such from our work in chapters 5 and 6. The experimental and validation efforts for both greatly benefited from the capabilities of the framework. They also illustrate that Splay can not only be used to write short prototypes, but also complex, long term running, applications.

Conciseness of implementation and ease of use of SPLAY allow it to be an educational tool of choice [108]. In a controlled environment (reliable network and nodes), it is possible to focus only on the algorithmic part of the problem. In more adverse conditions (natural or artificial, e.g., using churn system), SPLAY permits efficient error handling.

The name SPLAY is derived from "*sp*ontaneous over*lay*", i.e., a distributed application or overlay network that is instantiated for a specific task, whose nodes are usually deployed simultaneously, and whose behavior (protocol) and lifetime are decided at instantiation time.

1.3 Organization of the Thesis

The thesis is organized as follows.

In Chapter 2 (Context), we introduce distributed systems, their evolution, their main characteristics, how they are used by enterprises and why they are complex to design. In this context, we analyze more in depth some representative distributed systems and explain where and how could SPLAY have helped in their design. Finally, we give an overview of alternative technologies.

In Chapter 3 (The SPLAY Framework), we present all the fundamentals SPLAY components and explain the main design choices. We provide details about the Lua language, the sandbox, how to write SPLAY applications, and how to do distributed experiments using SPLAY.

In Chapter 4 (Evaluation of SPLAY), we perform a thorough evaluation of SPLAY performance and capabilities. We study the performance of SPLAY by implementing real distributed systems and reproduce experiments that are commonly used in evaluations to compare it to other widely-used implementations. We demonstrate the usefulness and benefits of SPLAY rather than evaluate the distributed applications themselves.

In order to demonstrate the benefits of distributed system research using SPLAY, the next chapters will present two systems that have been almost entirely implemented and evaluated using the SPLAY framework.

In Chapter 5 (PULP), we present PULP, a generic and efficient push-pull dissemination protocol.

In Chapter 6 (Collaborative Ranking and Profiling), we present a collaborative search companion that collect user searches to build profiling information over time and enhance search experience by providing tailored results.

We summarize, conclude and give an outlook of future work in Chapter 7 (Conclusion).

Chapter 2

Context

Every cloud engenders not a storm.

William Shakespeare, "Henry VI"

2.1 The Advent of Distributed Systems

2.1.1 Mainframe computers

Mainframe computers are the oldest form of generalized computing. They first appeared in enterprises during the 1950s. For nearly three decades, they were the only form of computing available, long time before the apparition of personal computers. Today, they still play a major role in the world's largest corporations that have done a major investment in their application and data.

The incredible success of mainframes is mostly related to remarkable compatibility that permits to combine application written "yesterday" with applications written more than fifty years ago.

Mainframe computers are an emblematic example of vertical scalability. While there are some distributed mechanisms available for modern mainframe, the essence of this architecture is to appear as a whole, indivisible system where each component can be replaced or updated transparently.

They way of achieving fault tolerance and availability is also emblematic: an important part of mainframe computer's hardware and software is dedicated to failure detection of components and to have the ability to replace them without affecting normal operations (hot swap, spare components).

Despite their quality, these systems have some limitations. First, vertical scalability is always bounded to an upper limit defined by the highest end components a mainframe features and the maximum number of these. Second, the cost of a mainframe computer limits its adoption to businesses that can afford a very high cost per operation (banks, insurances, ...). Third, mainframes are mostly based on proprietary hardware, operating system and applications. Interoperability is very limited, the customer is tightly coupled with a vendor, its architecture, its products, its support and its prices.

If mainframe computers still have a bright future, in corporations were storing and manipulating sensitive data in-house and with extreme reliability is a major requirement, most companies are, however, interested in limiting their operational costs. For that purpose, they can deploy complementary systems doing data crunching at a more affordable, or on-demand, price.

2.1.2 Early years of Internet

Both centralized and distributed paradigms have been part of the computer networks since their early years. A good example of this symbiosis is the email system. The system itself is a distributed system where each server is responsible for a specific domain and acts as a relay when receiving emails for other domains. When users of a specific domain connect to the corresponding server to retrieve their emails, the server acts as a centralized (client-server) system.

In 1993, the World Wide Web (commonly known as the Web) become publicly available. As the name suggests, the brilliant idea behind the Web is to easily connect different resources (pages from the same and different domains) together by using text links (hypertext [22]). Before this invention, exchanges between users were generally limited to people sharing the same interests (newsgroups, mailing lists, etc.), mostly in academic environments where Internet was already available.

The Web immediately permits enterprises to publish informations for everyone by just having a domain name and a server. The first Web browser (MOSAIC) provided a simple access to the Web and was at the origin of the public interest for the Internet.

Until the mid-nineties, the main services provided by the Internet were static Web pages, emails, newsgroups, FTP (File Transfer Protocol) and IRC (Internet Relay Chat). The number of users was small (but growing fast) and home connections without broadband access: textual content was favored over images. Serving static Web pages was not very resource-consuming and was generally provided by a single server with just some extra hardware resources if needed (vertical scalability). The content being static, simple caching strategies were widely used by providers to reduce the traffic outside their network (using transparent or non transparent proxies).

2.1.3 Commercial Web Advent

The advent of commercial Internet websites (e-commerce) greatly changed the situation. Due to their dynamic contents and to the fact that they must keep sessions containing data from the current visitor (shopping cart, etc.) and query external system (credit card, etc.), such systems needed much more resources for their operation.

Henceforth, services provided over the Internet becoming more and more complex, vertical scalability was not enough anymore to support them.

To scale, different strategies were applied. The static content can be served very efficiently with almost no CPU requirements and benefits from the caching mechanisms of each ISPs (Internet Service Provider). The dynamic content can be split in independent parts being processed by different servers. Most dynamic content can also have a certain TTL (Time To Live) and thus can also be cached on the server side. The pages where then generated by multiple load-balanced servers, each of them asking shared data from a database server. In this scheme, the central database often becomes the bottleneck of the system.

At that time, there was the need to put the servers near the customers. The main problem was not the delay (theoretical delay is lower than 1/10th second between any two points on earth [1]), but the bandwidth bottleneck between different regions of the globe [2]. These various instances often need to be synchronized, at least partially, for example to have global user accounts. This leads to new difficulties and consistency problems.

[1] The distance between two antipodes is half of Earth's circumference so $\sim$ 20000 km. If the speed of light in vacuum is c, wave propagation in optical fibers is $\sim 2/3c$, while in copper wire, the speed generally ranges from $.59c$ to $.77c$. Obviously the physical links are not deployed in a straight line, many repeaters are on their route, and routing rules are not always optimal.

[2] The first trans-ocean submarine Internet cables was set up in 1995 [2].

2.1.4 Google: Scaling with the Web

In 1998, Google, launched his new search engine. The success of Google was essentially based on their new page relevance algorithm, PageRank [31], but that was not their only innovation.

To parse the massive amount of data gathered by their Web crawlers, Google has been the first to build a server farm based on cheap commodity hardware; instead of spending a lot of money for highly priced servers, they opted to add more computers with cheaper hardware and manage fault tolerance at software level using their own distributed system.

Their system was able to do an efficient reverse indexation (word based index) and return the most relevant results (sorted by PageRank) to a search automatically whereas previously there were only manually generated directories.

It is probably one of the first successes of a new kind of distributed infrastructure able to replace the classical super computer architecture. The most important feature was that Google's architecture could easily scale by just adding new nodes as the number of Web pages was growing (approximately one billion pages at the end of the nineties).

2.1.5 Big Data Challenge

Since the beginning of the 21st century, we observe an exponential growth in data generated by the Web.

Globally, the involvement of users has evolved from spectators, only consuming data, to actors, generating much data by themselves.

It all began with the advent of blogs and CMS (Content Management Systems). The technical barrier for publication has been lowered by these user-friendly technologies and every user had finally the possibility to create his own website and add content to it.

More recently, social networks with AJAX (Asynchronous Javascript and XML) Web content, encourage this editorial practice. Everybody is asked to participate and to share more and more personal information. This evolution is commonly called *Web 2.0*.

The quantity of data to process has grown faster than the computing power of each individual computer. Moreover, one cannot simply replace a set of computing resources by a new more powerful one. The availability of the services must be ensured, and scalability must be achieved by adding resources to the existing set.

With the Internet, any startup can expect to see the number of its users growing by several orders of magnitude within a few years. Such a change implies to particularly focus on the problematics of scaling. This is the best way to be able to keep a stable architecture while tolerating large computing variations without making a disproportionate investments in resources that are not yet used. The availability of resources in the cloud is the perfect complement to this kind of architecture.

2.1.6 Mobile Computing

Mobile applications, and more generally wearable computing, have completely changed the way users interact with Internet. Before, a user has to explicitly consult a service to get data from it (pull). Today, mobile users are *subscribed* to services (having a mobile application is exactly as being subscribed to a service) and receive updates or notifications without requesting them (push). More than that, most users are unaware of the data they are publishing all the time: their location (GPS), usage statistics, etc.

The amount of data is no more related to the number of users explicitly publishing something, but directly proportional to the number of users subscribed to a particular service.

A smartphone can also automatically upload/publish a picture a user has just taken on social networks or a remote storage while automatically adding metadata (GPS position, orientation, camera model, etc.) without any explicit user action. The amount and the exact type of information sent is unknown to most users.

2.2 Distributed Systems Characteristics

Distributed system are a set of computers (or more generally any device with a CPU, RAM, and, very often, persistent storage) connected to a network and globally managed via a dedicated software. Computers being part of the system are usually called *nodes* (sometimes *peers* when dealing with a decentralized system).

In recent years, many applications have been built upon the distributed computing model. Theses applications can potentially support a very high number of users connected to them or being an active part of the system itself (peer-to-peer). Very common examples of distributed systems are search engines who are using server farms to index the Web and to reply to queries, social websites like Twitter, Facebook and LinkedIn, dedicated applications like Skype and Zattoo or MMORPG (Massively Multiplayer Online Role-Playing Game) games like World of Warcraft.

Several types of distributed systems exist that vary according to the nature of the hardware and the type of software abstraction. We can typically classify them in two major categories: tightly coupled and loosely coupled systems. On tightly coupled systems, there is a global view of the system. The hardware is well known and stable. Computers are connected via a dedicated network and network availability is guaranteed. Failures can be corrected by the owner of the system. Typical example of this structure are computer grids and server farms.

On the other end, loosely coupled system have more diversity: computers can use different operating systems, be in different locations and use different hardware (CPU power, amount of RAM, disk space, bandwidth).

P2P (peer-to-peer) systems are mainly characterized by the fact that each peer acts both as a client and a server. Most P2P systems use the computer resources of the participating users and can also be considered as highly dynamic or volatile systems because the users (and their corresponding peers) stay online a limited and unpredictable amount of time, hampering making assumptions about their future availability.

All these systems need to communicate using message passing and not via shared memory, like local applications do. Some implementations hide this communication model by presenting the distributed system as a single system. A common example of this usage are distributed operating systems that appear to the users as a single computer [3] [4].

Other commonly used abstractions are RPCs (Remote Procedure Call) and distributed file systems. The first one permits to call functions of a remote application instance almost as easily as a local function call. The main difference is that the function may fail due to an external condition (network, remote node shutdown, etc.) and this particular situation must be detected. Distributed file systems provide a global common storage accessible by all the nodes. The problems in that case are related to synchronization and efficiency.

Enterprise distributed systems are an important class of LSDS (Large Scale Distributed Systems) with a large amount of nodes, where new nodes can be added when there is a need to scale. This is the opposite of a static distributed system like a cluster of computers or a grid where the amount of nodes is mostly fixed (although some cluster infrastructure support addition of hardware nodes on the fly).

2.2.1 P2P Systems

P2P systems are a specific form of distributed architectures where nodes act both as consumers and suppliers of resources. Users wanting to benefit from specific material provided by the P2P network need to run their own node to access them.

In a P2P network, tasks such as nodes organization, indexation or searching for files are shared amongst all the participating nodes. This requires each node to make available part of its computing resources (processing power, disk storage, network bandwidth) to other nodes. Finally, some user may provide some unique material to be shared in the network.

The hardware problems being delegated to the participants, the system as a whole cannot make any assumptions about the availability of nodes.

Some of the main challenges of P2P applications are:

Cumulative/recursive delays. Many nodes may be involved (sometimes recursively) to answer a request.

Persistence. If all nodes containing a unique piece of data disappear, this data will not be available until at least one of these nodes comes back.

Replication. To ensure persistence, data need to be replicated.

Privacy. Private data can be distributed (stored on many nodes) but only some of the nodes with the proper credentials, must be able to read it.

Slow nodes. Many algorithms involving a set of nodes (especially in a recursive fashion) perform badly with slow or unresponsive nodes.

Ill-behaved peers. Peers that do not play by the rules (selfish behavior, data corruption, etc.), shall not tamper the service availability.

NAT traversal. The lack of public IPv4 addresses has spread the usage of NAT routers in enterprise and family networks. Without additional routing rules, a server behind a NAT is not directly accessible. Several solutions exist to leverage this [70, 109], but they require complex mechanisms and may not deal with all possible NAT couplings.

Asymmetric bandwidth. Most broadband connections have different upload and download speed (download being favored). The asymmetry makes it difficult for a balanced sharing of resources.

Bootstrap. To access a P2P network it is necessary to know a node already participating (or someone that can give a list of participating nodes).

Most of these problems result from the lack of any guarantees on the participants and the uniformity of P2P networks.

To overcome these difficulties, hybrid systems are often used:

Permanent super nodes. Super nodes are elected nodes that get more responsibilities in the network. It is a common mechanism in P2P systems to improve the stability and efficiency of the network. Instead of relying on elected nodes, highly reliable permanent super nodes can be directly inserted by the network owners. Super nodes help the system to behave better, but they are not necessary for the network to operate.

Indexes/Trackers. Additional servers are used to keep track of participating nodes and index what they have. The exchanges between the nodes still use pure P2P. The external servers are not active peers of the system and use different software. The network cannot work without them.

2.2.2 Cloud Computing

Cloud computing provides the ability to access computing resources/applications, on demand via Internet. It is not a technological evolution (every involved technologies already existed before being deployed in the cloud) but an economical one.

Cloud computing offerings are generally divided into three categories:

IaaS (Infrastructure as a Service) Provides virtualized resources, in the form of virtualized computers with a specific amount of CPU, memory, disk and bandwidth.

PaaS (Platform as a Service) Provides an execution platform for applications, often providing fail-over mechanisms and load balancing.

SaaS (Software as a Service) Applications available via Internet. The user doesn't manage any part of the application, he just pays for access to it.

With cloud computing, even a very small company can launch a service without investing capital in heavy equipment. There are also economies of scale and ecological considerations in cloud computing. Companies share the same resources instead of investing in static resources that would be idle a significant part of the time (idle resources consume less, but still consume something) by using the same resources, but at different times (different peak hours in the same day depending of the time zone or different days in a year).

2.2.3 The CAP Theorem

Distributed systems are prone to issues that do not exist in monolithic systems: failure of a number of nodes, messages loss (loss of network connectivity, saturated bandwidth, etc.).

In this context, it is important to define more precisely which properties can be achieved and under what conditions.

The initial "CAP" conjecture was made by Eric A. Brewer in an invited talk [30] at PODC 2000. It was proven later by Gilbert and Lynch [57] as the CAP theorem.

The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three guarantees:

C (Consistency) All nodes see the same data at the same time.

A (Availability) Every request will receive a response.

P (Partition tolerance) The system will continue to operate, despite partial failures and message loss.

As described, a CP system appears to be an *unavailable*, partition tolerant and consistent system ! The misunderstanding resides in the meaning of each letter. A non-C system will have only casual consistency (consistency is not a goal). In a CP system, availability is only sacrificed in case of network partition.

Research in DBMS (DataBase Management Systems) has (mostly) focused on providing ACID guarantees:

A (Atomicity) All operations in a transaction will be completed or none will.

C (Consistency) The database will be in a valid state before and after each transaction.

I (Isolation) Each transaction will behave as if it was the unique operation on the database at the same moment.

D (Durability) The result of a complete transaction is stored permanently.

To provide better availability, graceful degradation and performance, "C" and "I" must be sacrificed. Brewer proposes the BASE properties (Basically Available, Soft-state, Eventual consistency) to replace the ACID ones. In these new databases, it is sufficient to be in an eventually consistent state rather than being consistent after each transaction.

Developing software using fault-tolerant BASE is harder than relying on ACID systems, but necessary to scale up. There is a continuum between ACID and BASE: one can choose how close one wants to be to either end depending of the application requirements.

2.3 Distributed System Design

Developing large-scale distributed applications is a highly complex, time-consuming and error-prone task.

One of the main difficulties stems from the lack of appropriate tool sets for quickly prototyping, deploying and evaluating algorithms in real conditions, when facing unpredictable communication and failure patterns.

Nonetheless, evaluation of distributed systems over real testbeds is highly desirable, as it is quite common to discover discrepancies between the expected behavior of an application as modeled or simulated and its actual behavior when deployed in a live network.

While there exist a number of experimental testbeds to address this demand (e.g., PlanetLab [1], ModelNet [127], or Emulab [131]), they are unfortunately not used as systematically as they should. Indeed, our firsthand experience has convinced us that it is far from straightforward to develop, deploy, execute and monitor applications for them and the learning curve is usually steep.

Technical difficulties are even higher when one wants to deploy an application on several testbeds (deployment scripts written for one testbed may not be directly reusable for another, e.g., between PlanetLab and ModelNet). As a side effect of these difficulties, the performance of an application can be greatly impacted by the technical quality of its implementation and the skills of the person who deploys it, overshadowing features of the underlying algorithms and making comparisons potentially unsound or irrelevant. More dramatically, the complexity of using existing testbeds discourages researchers, teachers, or more generally systems practitioners from fully exploiting these technologies.

Many other distributed deployment platforms could be leveraged for activities related to distributed system design and evaluation (research, teaching, monitoring, etc.). Examples of such platforms are networks of idle workstations in universities, research facilities and companies, networks of personal PCs whose users donate some idle CPU time or ubiquitous hardware like smart phones.

As any PlanetLab user can testify, it is a highly technical task to develop, deploy, execute and monitor applications for such complex testbeds. Technical difficulties are even much higher when one wants to develop for and deploy and execute applications onto several testbeds. The lack of adequate tools for these tasks increase significantly the amount of work to use them efficiently.

These various factors outline the need for novel development-deployment systems that would straightforwardly exploit existing testbeds and bridge the gap between algorithmic specifications and live systems.

For researchers, such a system would significantly shorten the delay experienced when moving from simulation to evaluation of large-scale distributed systems ("time-to-paper" gap). Teachers would use it to focus their lab work on the core of distributed programming—algorithms and protocols—and let students experience distributed systems implementation in real settings with little effort. Practitioners could easily validate their applications in the most adverse conditions.

There are already several systems to ease the development or deployment process of distributed applications. Tools like Mace [72] or P2 [89] assist the developer by generating code from a high-level description, but do not provide any facility for its deployment or

evaluation. Tools such as Plush [18] or Weevil [130] help for the deployment process, but are restricted to situations where the user has control over the nodes composing the testbed (i.e., the ability to run programs remotely using `ssh` or similar).

They are not meant for testbeds where only limited access rights are available, and/or where the owner of the nodes wants to strictly restrict resources usages, or wants to prevent any erroneous or malicious application harming the system.

2.4 LSDS in Industry

2.4.1 Introduction

To face the *Big Data Challenge*, LSDS have become a fundamental part of the enterprises computing ecosystem.

In order to present the LSDS goals and problems, we will analyse four typical LSDS building blocks, how they stack together and, later, explain where SPLAY could have been useful during their prototyping phase.

Considering Google has pioneered the domain [1], we will focus on their papers describing four of their production-ready LSDS implementations:

GFS (Google File System) A DFS (Distributed File System).

MapReduce A framework for parallel computation.

Chubby A distributed lock/file/DNS service.

BigTable A NOSQL (Not Only SQL) database.

We will also mention *The Hadoop Project* that provides open source software to build scalable and data-intensive distributed applications. Initially, it was mainly a Java implementation of architectures corresponding to those described in the MapReduce and GFS papers. Today, it also embraces many related distributed system projects that are built upon this underlying infrastructure (Cassandra, HBase, ZooKeeper, etc.).

[1] Their success is also often associated with the fact they were able to build distributed systems and algorithms that run on cheap commodity hardware instead of expensive servers or mainframes..

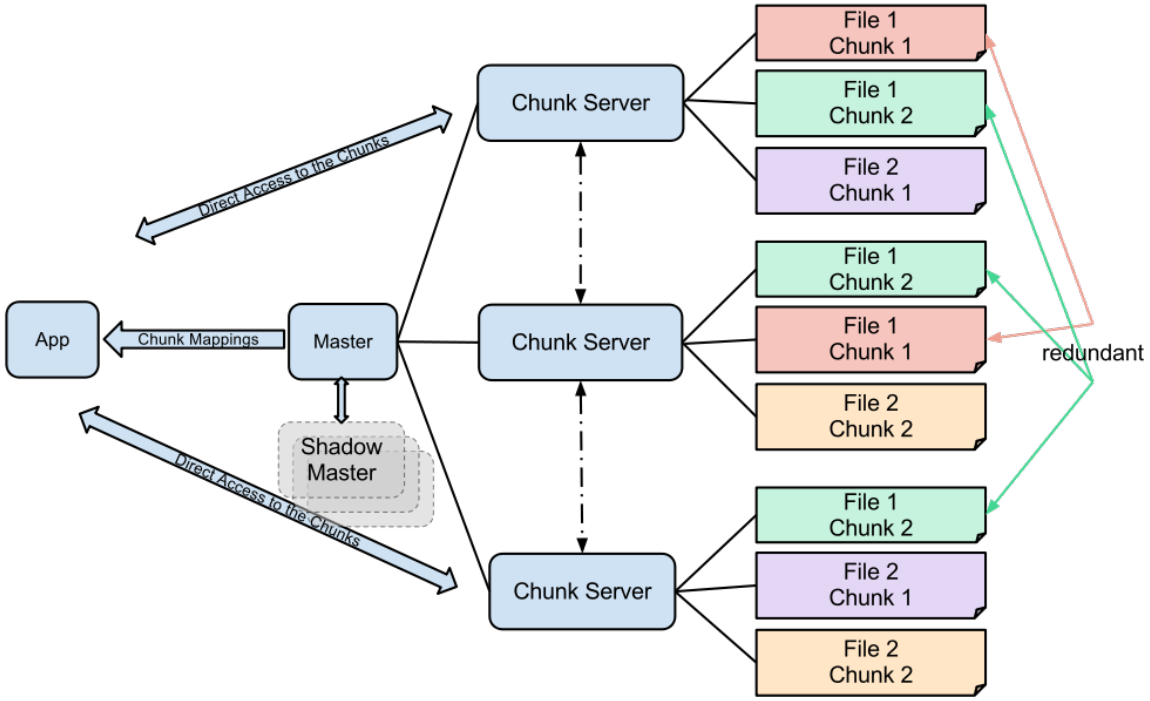


Figure 2.1 – GFS Architecture [Wikipedia, CC0 license]

2.4.2 GFS

GFS is a distributed file system. It was first described in [56]. HDFS [29] (Hadoop Distributed FS) is the corresponding Hadoop implementation.

It was developed with the following goals and assumptions:

- Provide redundant storage of large amounts of data using inexpensive computers (high failure rate).
- Optimize datacenter bandwidth by providing location aware replication.
- Millions of huge files ($>$ 100Mo, typically several GB).
- Write once (mostly), streaming reads.

The GFS cluster architecture consists of two types of nodes: one master node and a large number of chunk servers (see Figure 2.1). Chunks are multiple parts of a single file.

For efficiency reasons, each chunk has a fixed size of 64Mo and is assigned a unique 64-bit identifier by a master node at the time of its creation. The master node keeps the mapping between files and the constituent chunks and coordinates operations on files: it stores their metadata and select the chunk servers involved in the successive read or write operations. Finally, a GFS client library will permit to directly communicate with the master and the chunk servers. Basically, the chunk servers only know how to store, serve and delete chunks, without having any global view.

The master node is the interface to all operations between the clients and the physical storage (the chunk servers). As a central point of coordination, it will limit the scalability of the cluster. For that reason, the master involvement in file operations must be kept as low as possible. The data exchanges are done directly between chunk servers (chunk replications) and between the GFS clients and the chunk servers.

When the master receives a command to store a new file, it first creates an entry for the metadata information received from the GFS client: namespace, filename, type, size, times, etc. Then, the master will reply to the GFS client a selection of the less loaded chunk servers that will store the chunks. When each chunk is safely stored, the master will update the file metadata with the locations of the chunks and their checksums.

After a client request for a file, the master, knowing where chunks are stored, will return the list of chunks with their corresponding chunk servers (the master will reply an incomplete list of chunk servers for a given chunk, again it selects the most appropriate chunk servers depending of their load).

The master also organizes the replication of chunks. Three copies are the minimum (they ensure safety and availability of the data) but depending of the level of requests for a particular chunk, the master will dynamically increase the number of replicas to improve load balancing (and will recover some space when the number of requests decrease). The master permanently monitors chunk servers for their availability and trigger the replication process in case the replication degree of some chunks has fallen beneath a threshold.

In order to have a bandwidth effective replication, GFS is location aware (it knows on which switch the chunk server is connected, other chunk servers on the same switch are considered a part of the same *rack*. In a typical setup, the data are duplicated three times: two times in a close proximity (same rack, local switch) and once more remotely (in another rack). The bandwidth inside the same rack is probably higher, and in all cases cheaper.

When a file is deleted, the master will launch a garbage collection process that ensures that the corresponding chunks will be deleted.

To have the master be fault tolerant, several *shadow* masters run in parallel of the actually used master. They replicate all its operations using a replication log. If the master goes down, a new master is automatically elected and replaces the failed one.

As previously stated, the role of the chunk servers is much simpler: they are mainly used to store and serve chunks when requested. Additionally they also checksum the chunks when writing them to ensure correctness. The limited size of the chunks ensures a good repartition between chunk servers and also improves memory usage (small chunks permit to better fill the available memory of each server).

The application communicates with the distributed file system using the GFS client. The GFS client does not act as a simple library, but is an active component that first communicates with the master using RPC, then with the chunk servers for the data stream. The GFS client also caches the master's files metadata to offload it as much as possible.

Some of the major implementation challenges in GFS include:

- Effective replication of chunks.
- Master recovery: replicated logs of operation, shadow master with replica.
- File mutations (write/append):
    - Replica must be synchronized.
    - Master involvement must be minimized.
- Load balancing read/write operations for chunk servers.

A typical GFS cluster involves hundred of nodes running in a well-defined environment: all chunk servers are very similar (CPU, storage, RAM) and the bandwidth between nodes is known.

2.4.3 MapReduce

MapReduce is a framework for parallel computations. It was first described in [44].

The MapReduce algorithm is directly inspired by two common operations used in functional programming to work with lists:

Map Applies a function to all elements of a list and return a new list.

Reduce High-order function, also called a *fold* function in functional programming, that recursively analyzes a data structure and recombines it, returning a result (if the result is a list, it can be given as an input to another *map* or *reduce* function).

The MapReduce framework provides a parallel execution environment for every problem expressed in the form of a *map* and a *reduce* functions.

The initial data domain is the raw input. It is then split into chunks represented by pairs *(k1, v1)*. If the input is a file system, *k1* can be the filename and *v1* the content. If the input is a video stream, *v1* can be the content of the chunks while *k1* being the hash of the respective chunks. The *(k1, v1)* pairs are then dispatched to the mappers.

Each mapper will then compute $Map(k1, v1) \rightarrow list(k2, v2)$. *(k2, v2)* is a new data domain were we search a solution for each *k2*. *k2* will also be used to select the reducer node (the node that will apply the reduce function) typically using a classical *k2* mod *number_of_mappers* dispatching (it is fundamental that each identical *k2* goes to the same reducer).

The reducer node for *k2* will receive all the corresponding *v2*. It can then perform the reduction $Reduce(k2, list(v2)) \rightarrow k2\_solution$.

MapReduce is a very smart way to parallelize operations on massive data inputs. But, as always, these distributed operations can be more or less effective depending of the quality of the architecture's implementation.

A very challenging part, in this case, is to provide data to the mappers and reducers. To effectively compute *Map(k1, v1)* and *Reduce(k2, list(v2))*, the nodes have to get the corresponding data locally.

The data chunks (*v1*) should not be too small or too big: small enough to fit into the mapper's memory (avoiding swapping) and to provide a good load balanced distribution between all mappers, big enough to not add unnecessary overhead.

The reducers need to wait for all the *v2* values associated to one *k2* (the mapping process must be finished) before doing the reduction (generally, *v2* values are also ordered by an additional application specific function before being reduced). A badly balanced mapping step would delay the beginning of reductions.

2.4.4 Chubby

Chubby [33] is a lock service for loosely-coupled distributed systems. It provides coarse-grained locking as well as a reliable distributed file system. The design emphasis is on availability and reliability (through replication), as opposed to high performance.

Chubby is now used in key parts of the Google's infrastructure, including GFS, MapReduce and BigTable. Google uses Chubby for distributed coordination and also to store small amounts of metadata.

A typical Chubby cell consists of five replicas (up to two servers can be down), each running on a dedicated machine. At any time, one of this replica is considered to be the *master*. When a Chubby client contacts one of the replica not being the master, the replica replies with the master's network address. If the master fails, a new master is automatically elected. The asynchronous consensus is solved by the Paxos [77] [76] protocol.

Chubby exposes its data using paths similar to the UNIX file system. Access to files is filtered using ACLs (also stored in the file system) and a process can request a lock before doing operations on a file. The locks are advisory and not mandatory: the lock mechanism is only applied between processes requesting a lock; non-locking access to a file is always possible.

Chubby uses the Paxos protocol to synchronize a log file describing all successive operations in the database. Replicas agree on the execution order of client requests using the consensus algorithm. When the consensus is reached, the new value is append to the log file. The value is then written in the local database to be used by the Chubby APIs.

As stressed in [38], implementing theoretically well studied protocols such as Paxos is much harder than expected. Authors emphasize the difference between a pseudo code algorithm and a production-ready system where many features and optimization must be done before being usable.

In the purpose of validating their implementations, Google engineers have developed a lot of stress tests, error injections, etc. Some of these failure/recovery mechanisms were very dependent on the underlying OS (thread scheduling, disk synchronization, etc.).

Hadoop's "equivalent" for Chubby is ZooKeeper [59]. However, the design and features of ZooKeeper slightly differ from the ones of Chubby (e.g., ZooKeeper supports notifications and a tree structure for storing shared metadata). ZooKeeper does not use Paxos, but Zab [64, 65], , which provides linearizability through primary-backup replication instead of serializability in order to improve read scalability.

2.4.5 BigTable

BigTable [39] is a NOSQL column-oriented database (values for a column are stored contiguously) designed to scale to a very large size: thousands of server storing petabytes of data with high performance, scalability and reliability. Google uses BigTable to store data for more than sixty of their products. The corresponding Hadoop implementation is HBase.

NOSQL databases have been created to deal with huge quantities of data that the traditional RDBMS (Relational DataBase Management System) solutions could not cope with.

NOSQL is a class of DBMS that does not follow the widely used relational model:

- The query language is not SQL.
- It may not provide ACID guarantees.
- The architecture is distributed and fault tolerant.

A BigTable is a sparse, distributed, persistent, multi-dimensional, sorted map.

Sparse. Few entries are defined compared to the total index space.

Distributed. The implementation provides fault-tolerance and availability.

Multi-dimensional. The map values are indexed by *(row, column, time)*. Time acts as a revision control system.

Persistent. Data will not be deliberately or inadvertently deleted by the system.

Rows keys (arbitrary string up to 64k) are lexically indexed and distributed across servers. Read/write of a row is atomic for all its content. A range of rows, called a *tablet*, is the unit of distribution and load balancing. Each table server contains many tablets, so all the operations concerning a row can be done locally on the server that owns it. The tablet size is limited to approximatively 200MB; when it grows, the tablet is split.

Data items are often primarily indexed by their URLs (in reverse domain order). Different URLs from the same domain will be lexically close, thus insuring that pages of a domain are stored near each other (in the same tablet and/or the same server).

As each column needs to be part of a column family, the key will be in the form: 'family:qualifier'. The families serve as access control and also as compression groups. The content of a family generally being of a same type, compression gains are better (and the chosen compression algorithm can be set for a particular family type).

The timestamp index (in microseconds) serves as a version control. By default, using only row and column indexes, the last content is returned (or written using the actual time). The timestamp is also used by the garbage collector to keep the latest n revisions or keep the revisions where their age is smaller than a limit.

The tablets are stored on GFS using the *SSTable* (Sorted String Table) file format. The SSTables are immutable, contain a number of arbitrary sorted key-value pairs inside and an index. An SSTable completely or partially (just the index) mapped in memory is called a MemTable. Writes goes directly in the MemTable, reads check first the MemTable then the SSTable indexes. Periodically the MemTable is flushed on disk as an SSTable. Periodically the SSTable are merged.

A BigTable master is responsible for garbage collecting unused SSTables, monitoring the availability of tablet servers, load balancing the tablets between servers and organizing the recovery of tablets when a server is failed (the master will assign its tablets to new servers and prepare the recovery process). Multiple master servers are available, but only one is active, Chubby is used to obtain the lease.

To find which server manage a specific table entry, a two level meta-tables indexes the locations of tablets. The root meta-table location is also stored in Chubby.

2.4.6 SPLAY Applicability

In the following sections, we emphasize the role that SPLAY could have had in the design of the previously seen distributed systems. Compared to most other tools, SPLAY permits researchers to evaluate prototype applications on conditions very similar to the final ones (e.g., using similar hardware and network architecture). Even if SPLAY will not help for the final tuning (or operating system specific tuning), its performances permit us in most cases to have implementation that can handle a load comparable to that of the final application and save a lot of time during the conception phase until the stabilization of the protocols.

2.4.6.1 GFS

To evaluate a similar environment in SPLAY we could have used a local cluster of computers. Each computer would act as a rack containing the chunk servers (represented by a splayd [1] instance running on the computer). The chunk servers running on the same computer will have access to a very high bandwidth that can be limited by an external tool like *trickle* because SPLAY currently does not include integrated bandwidth shaping. The connectivity between the nodes of the cluster will represent the connectivity between the racks in the GFS datacenter.

The master node (and shadow masters) can be placed on a dedicated computer as if they were not on the same *rack* as chunk servers.

SPLAY makes it very easy to exchange control commands between the nodes using the integrated RPC mechanism. For data transfers, plain TCP connections can be used.

SPLAY provides a file system abstraction to store files (in this case, chunks) on the underlying file system with a very low overhead. The available disk space can be split between all the splayd running on the same computer. To checksum each 64KB blocks of chunks, SPLAY provides the MD5 or SHA mechanisms directly from OpenSSL C libraries.

2.4.6.2 MapReduce

A production-ready MapReduce is tightly coupled with GFS. Even if SPLAY permits us to deploy multiple applications (one for GFS and one for MapReduce) at the same time and make them communicates, we probably would be too far from the real system to provide a realistic implementation at a performance and scalability point of view.

[1] A splayd is a node in the SPLAY architecture (see Chapter 3).

In that case, at least two alternative approaches could be considered:

1. Bundle the production ready GFS client with splayd: that way we could test the MapReduce application on top of a realistic layer.
2. Completely skip the DFS part to concentrate on a more unified MapReduce-only approach.

The unified approach, in this case, consists at sending the data blocks (*(k1, v1)* and *(k2, v2)*) directly between the nodes (not using a DFS at all). Each node will have its own role (input, map, reduce, sort) and will queue the job on its local file system before being processed.

As splayd internally uses only coroutines and no threads, it will at most use only one CPU core. When using SPLAY for computational tests, one will run (at least) as many splayds as cores present in the physical machine. Map and reduce being functions that can possibly use a parallel implementation, the SPLAY approximation will be to run multiple independent processes instead of a parallelized one.

In this scenario, we could have each deployment validate one specific algorithm, but we can certainly do it better using the fact that SPLAY, using the Lua language, can directly pass code between nodes. Therefore a client could use the MapReduce test network for directly sending *(destination, data, map function, reduce function, sort function)* to an input node and get the result back (the Lua functions being transmitted between nodes along *(k1, v1)* and *(k2, v2)*, the overhead will often be negligible).

2.4.6.3 Chubby

While SPLAY cannot replace the testing of a production-ready implementation, we believe that it is an excellent candidate for implementing and testing algorithms like Paxos, which act as fundamental primitives for the upper layer (database and APIs).

Initially, much time is spent to just transform the pseudo code into a working protocol, before adding the needed features and optimizations. Using SPLAY RPCs will probably highly reduce the implementation time while producing a fully testable result.

SPLAY churn module and RPC dropping mechanisms can be very helpful to test failures and recovery in consensus protocols. In order to be compatible with the Paxos assumption on a persistent storage we would have to add support for SPLAY to preserve the node disk storage between shutdown and restart.

2.5 Related Work

SPLAY shares similarities with a large body of work in the area of concurrent and distributed systems. We only present systems that are closely related to our approach.

2.5.1 Development tools

On the one hand, a set of new languages and libraries have been proposed to ease and speed up the development process of distributed applications.

Mace [72] is a toolkit that provides a wide set of tools and libraries to develop distributed applications using an event-driven approach. Mace defines a grammar to specify finite state machines, which are then compiled to C++ code, implementing the event loop, timers, state transitions, and message handling. The generated code is platform-dependent: this can prove to be a constraint in heterogeneous environments. Mace focuses on application development and provides good performance results, but it does not provide any built-in facility for deploying or observing the generated distributed application.

P2 [89] uses a declarative logic language named OverLog to express overlays in a compact form by specifying data flows between nodes, using logical rules. While the resulting overlay descriptions are very succinct, specifications in P2 are not natural to most network programmers (programs are largely composed of table declaration statements and rules) and produce applications that are not very efficient. Similarly to Mace, P2 does not provide any support for deploying or monitoring applications: the user has to write his/her own scripts and tools.

Other domain-specific languages have been proposed for distributed systems development. In RTAG [19], protocols are specified as a context-free grammar. Incoming messages trigger reduction of the rules, which express the sequence of events allowed by the protocol. Morpheus [15] and Prolac [74] target network protocols development. All these systems share the goal of SPLAY to provide easily readable yet efficient implementations, but are restricted to developing low-level network protocols, while SPLAY targets a broader range of distributed systems.

2.5.2 Deployment tools

On the other hand, several tools have been proposed to provide runtime facilities for distributed applications developers by easing the deployment and monitoring phase.

Neko [125] is a set of libraries that abstract the network substrate for Java programs. A program that uses Neko can be executed without modifications either in simulations or in a real network, similarly to the NEST testbed [46]. Neko addresses simple deployment issues, by using daemons on distant nodes to launch the virtual machines (JVMs). Nonetheless, Neko's network library has been designed for simplicity rather than efficiency (as a result of using Java's RMI), provides no isolation of deployed programs, and does not have built-in support for monitoring. This restricts its usage to controlled settings and small-scale experiments.

Plush [18] is a set of tools for automatic deployment and monitoring of applications on large-scale testbeds such as PlanetLab [1]. Applications can be remotely compiled from source code on the target nodes. Similarly to Neko and SPLAY, Plush uses a set of application controllers (daemons) that run on each node of the system, and a centralized controller is responsible for managing the execution of the distributed application.

Along the same line, Weevil [130] automates the creation of deployment scripts. A set of models is provided by the user to describe the experiment. An interesting feature of Weevil lies in its ability to replay a distributed workload (such as a set of request for a distributed middleware infrastructure). These inputs can either be synthetically generated, or recorded from a previous run or simulation. The deployment phase does not include any node selection mechanism: the set of nodes and the mapping of application instances to these nodes must be provided by the user. The created scripts allow deployment and removal of the application, as well as the retrieval of outputs at the end of an experiment.

Plush and Weevil share a set of limitations that make them unsuitable for our goals. First, and most importantly, these systems propose high-end features for experienced users on experimental platforms such as PlanetLab, but cannot provide resource isolation due to their script-based nature. This restricts their usage to controlled testbeds, i.e., platform on which the user has been granted some access rights. Second, they manage applications at a coarse grain, much like a shell script does, whereas SPLAY aims to control or observe the internals of the program under deployment. Last, they do not provide any management of the dynamics (churn) of the system, despite its recognized usefulness for distributed system evaluation.

2.5.3 Testbeds

Conjointly, a set of experimental platforms, hereafter denoted as *testbeds*, have been built and proposed to the community. These testbeds are complementary to the languages and deployment systems presented in the first part of this section: they are the *medium* on which these tools operate.

Distributed simulation platforms such as WiDS [86] allow developers to run their application on top of an event-based network simulation layer. Distributed simulation is known to scale poorly, due to the high load of synchronization between nodes of the testbed hosting communicating processes. WiDS alleviates this limitation by relaxing the synchronization model between processes on distinct nodes. Nonetheless, event-based simulation testbeds such as WiDS do not provide mechanisms to deploy or manage the distributed application under test.

Network emulators such as Emulab [131], ModelNet [127], FlexLab [107] or P2PLab [100] can reproduce some of the characteristics of a networked environment: delays, bandwidth, packet drop, etc. They basically allow users to evaluate unmodified applications across various network models. Applications are typically deployed in a local-area cluster and all communications are routed through some proxy node(s), which emulate the topology. Each machine in the cluster can host several end-nodes from the emulated topology.

The PlanetLab [1] testbed (and forks such as Everlab [62]) allows experimenting in live networks by hosting applications on a large set of geographically dispersed hosts. It is a very valuable infrastructure for testing distributed applications in the most adverse conditions.

SPLAY is designed to complement these systems. Testbeds are useful, but often, complex platforms. They require the user to know how to deploy applications, to have a good understanding of the target topology, and to be able to properly configure the environment for executing his/her application (for instance, one needs to use a specific library to override the IP address used by the application in a ModelNet cluster). In PlanetLab, it is time-consuming and error-prone to choose a set of non-overloaded nodes on which to test the application, to deploy and launch the program, and to retrieve the results. Finally, considering mixed deployments that use several testbeds at the same time for a single experiment would require to write even much more complex scripts (e.g., taking into account problems such as port range forwarding). With SPLAY, as soon as the administrator who deployed the infrastructure has set up the network, using a complex testbed is as straightforward for the user as running an application on a local machine.

Chapter 3

The SPLAY Framework

> If I had asked people what they wanted, they would have said faster horses.
>
> Henry Ford

In this chapter we present in details the architecture of the SPLAY framework. We specifically describe its main components, its programming language, its libraries, its security model and tools.

3.1 Overview

The SPLAY framework consists of about 15,000 lines of code written in C, Lua, Ruby, and SQL, plus some third-party support libraries. Roughly speaking, the architecture is made of three major components. These components are depicted in Figure 3.1.

• The controller, splayctl, is a trusted entity that manage splayds, controls the deployment and execution of applications.

• A lightweight **daemon** process, splayd, runs on every machine of the testbed. A splayd instantiates, stops, and monitors SPLAY applications when instructed by the controller.

• SPLAY **applications** execute in sandboxed processes forked by splayd daemons on

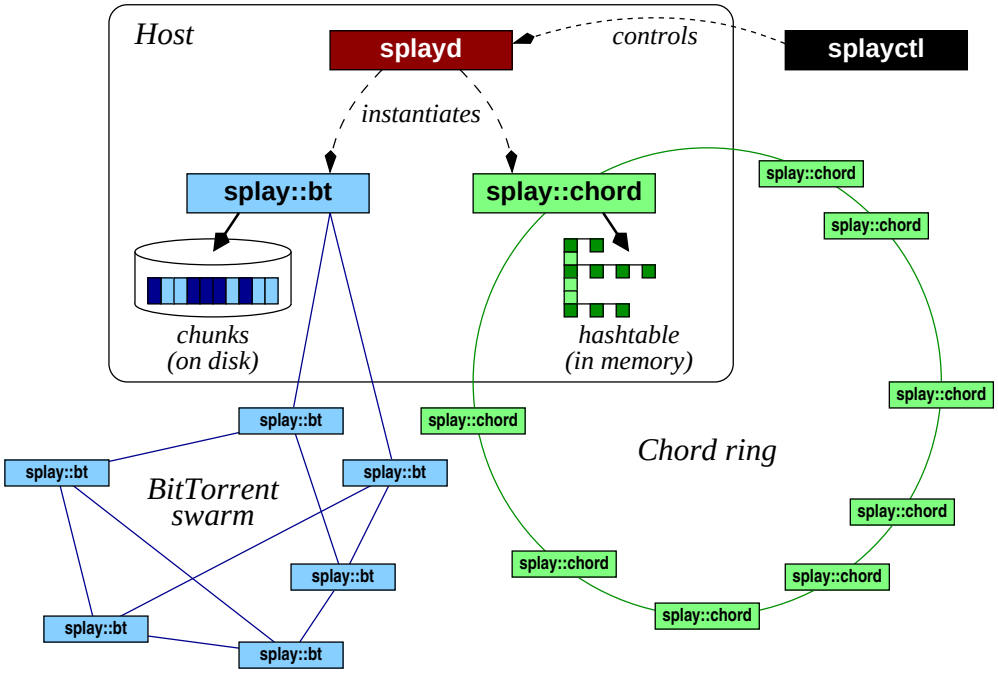


Figure 3.1 – An illustration of two SPLAY applications (BitTorrent and Chord) at runtime.

participating hosts. The package containing both the application and the description of how the execution must be done is referred as a *job*.

Many SPLAY applications can run simultaneously on the same host. The testbed can be used transparently by multiple users deploying different applications on overlapping sets of nodes, unless the controller has been configured for a single-user testbed. Two SPLAY applications on the same node are unaware of each other. They cannot exchange data via the file system and must communicate by message passing as for remote processes. Figure 3.1 illustrates the deployment of multiple applications with a host participating to both a Chord DHT and a BitTorrent swarm.

An important point is that SPLAY applications can be run locally with no modification to their code, while still using all libraries and language features proposed by SPLAY. Users can simply and fastly debug and test their programs locally, prior to deployment.

Sections 3.2 and 3.3 describe the internals of two major SPLAY components, respectively splayctl and splayd. In section 3.4, we will see how the deployment of a SPLAY application is done and the interactions between splayctl and splayds. In section 3.4 3.5, we will see the core of a SPLAY application: the Lua language, the scheduler and

the libraries. Then, in section 3.6, we will detail how RPCs are implemented and their usage.

In section 3.7, we put everything together to show some real applications that can be written with the framework. In section 3.8 we describe churn management. Finally, in section 3.9, we will talk about sandboxing design choices.

3.2 Controller

The controller plays an essential role in our system. It is implemented as a set of cooperating processes and executes on one or several trusted servers. The only central component is a database that stores all data pertaining to participating hosts and applications.

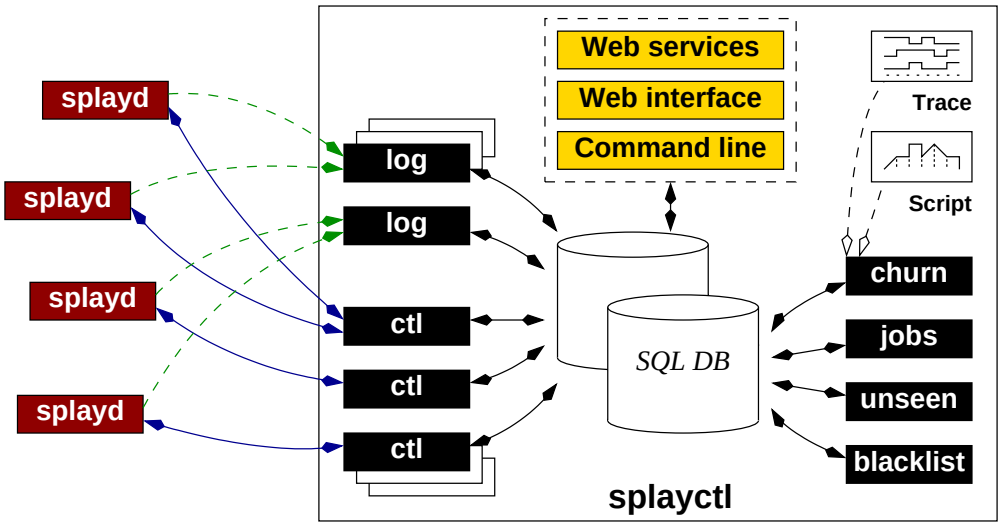


Figure 3.2 – Architecture of the SPLAY controller (note that all components may be distributed on different machines).

The controller (see Figure 3.2) keeps track of all active SPLAY daemons and applications in the system. Upon startup, a splayd initiates a secure connection (SSL) to a `ctl` process. Afterwards, the splayd will also open a connection to a `log` process for each running application.

The actual Ruby implementation permits us to manage several thousands of splayds using a multi processes architecture on one server. This multi-process architecture was

initially the result of the lack of native threads in Ruby 1.8 and the poor performances of its internal threading system. Now, this architecture also permits us to scale easily by increasing the number of servers and distributing the processes.

The deployment of a distributed application is achieved by submitting a job through a command-line or Web-based interface. These interfaces also permits us to monitor the status of the running jobs and access the collected logs. SPLAY also provides a Web services API that can be used by other projects.

The nodes participating in the deployment can be specified explicitly as a list of hosts, or one can simply indicate the number of nodes on which deployment has to take place, regardless of their identity. One can also specify requirements in terms of resources that must be available at the participating nodes (e.g., bandwidth) or in terms of geographical location (e.g., nodes in a specific country or within a given distance from some position). Incremental deployment, i.e., adding nodes at different times, can be performed using several jobs or with the churn manager. More information about the deployment are available in section 3.4.

Each daemon and job are associated with records in the database that store information about the applications and active hosts running them (or scheduled for later execution). The controller monitors the daemons and uses a session mechanism to tolerate short-term disconnections (i.e., a daemon is considered alive if it shows activity at least once during a given time period). Only after a long-term disconnection (typically one hour) does the controller reset the status of the daemon and clean up the associated entries in the database.

Several controller processes monitor one or more tables and react to specific events (modifications in the tables). More details about how each process works are given in the next section.

The controller actually does not manage directly users access rights, usage quota and external splayds registration. These features can be added through additional interfaces that directly act on specific controller's tables. [1] One implementation of this external interface that brings some of these features is the web-based interface SPLAYWEB. SPLAYWEB manages users, permits them to register their own splayds and to run their jobs using a web interface. It also provides some rudimentary web services APIs to access these features from a third party application.

[1] It would be better to provide a low level controller API for that in the future, rather than giving a direct access to the database.

3.2.1 Controller's Processes

As previously explained, the controller architecture is build of several cooperating processes. This permits us to achieve a very good scalability, separation of concern and easier management (e.g., to restart some processes individually).

Before seeing each processes more in details, it is important to give some insights about the database structure and tables that are used to store the data.

First the *splayds* table stores all the attributes and configuration of a splayd. This table is primarily used to select splayds using various criteria when a new job is submitted.

The *splayd_availabilities* table contains the history of the availability of each splayd. Using this table we can measure the average availability of each splayd and filter them using that criteria (e.g., to avoid running a long experiment on unstable nodes).

The *jobs* table contains all jobs submitted. When a job is deployed, the table *splayd_jobs* will contain the list of the splayds where the job has been deployed.

Finally, the *actions* table is used as a queue to send commands to the splayds.

ctl processes

When the controller starts, a pool of `ctl` processes are launched and during the registration part, each splayd will receive the instructions to connect to one of them.

After the communication initialization and state synchronization, the `ctl` process will monitor the *actions* queue (queue of commands for the connected splayd), send them and finally update the local splayd status in the database after having received the command's reply.

jobs processes

The `jobs` process dequeues jobs from the database and searches for a set of splayds matching the constraints specified by the user. In case the job needs to run under churn, an additional `churn` process is created to monitor the states of the running splayds and to switch them on or off during the time of the experiment.

This process has the complex task to find enough splayds meeting the job's criteria and to deploy it on them.

The *splayd_selections* table will be used to store the initial selection of splayds as well as the final list. Another table, *job_mandatory_splayds*, is used when submitting a job that absolutely must includes some specific splayds (by default, the controller chooses the most responsive, less loaded splayds).

log processes

As for the `ctl` processes, a pool of `log` processes are launched.

Each splayd will receive during the registration protocol the instructions to connect to one of them. The accesses to the `log` are restricted by several security checks, as is the total log size that can be received by a single splayd.

The cumulative output of all splayds running the job is stored in a plain text file named using the job's reference.

The role of `log` is to collect information about or from running applications. This information can then be processed to debug algorithms, analyze their performance, gather runtime statistics, etc.

other processes

The `unseen` process will monitor the state of splayds by sending them a regular ping to verify their availability (the ping will be added as a command in the *actions* table).

The `blacklist` process is responsible to ensure that all running splayds have the most up to date blacklist of hosts. Support for blacklisting has been added to provide protection against malicious SPLAY usage and if complains about someone receiving unwanted connections from one or more splayds are received. In this case, all successive attempts will be denied (before taking other measures).

3.3 splayd

The SPLAY daemon, called splayd, is the substrate to efficiently deploy a SPLAY application.

Splayd is a key component of SPLAY experiments as it manages the execution of one or more jobs and all the environmental aspects around that execution, however, using

splayd is not necessary to run a SPLAY application: the SPLAY libraries can be used standalone.

SPLAY daemons are installed on participating hosts by a local user or administrator. Ideally, they should be automatically started at boot time (e.g., by an *init.d* script on Unix). The local administrator can configure the daemon via a configuration file (see Listing 3.1), specifying various instance parameters (e.g., daemon name, access key, etc.) and restrictions on the resources available for SPLAY applications. These restrictions encompass memory, network, and disk usage. If an application exceeds these limitations, it is killed (memory usage) or I/O operations fail (disk or network usage). The controller can specify stricter—but not weaker—restrictions at deployment time.

```
1  splayd.settings.key = "local" -- received during registration
2
3  splayd.settings.name = "name"
4
5  splayd.settings.controller.ip = "localhost"
6  splayd.settings.controller.port = 11000
7
8  -- All sizes are in bytes
9  splayd.settings.job.max_number = 16
10 splayd.settings.job.max_mem = 12 * 1024 * 1024 -- 12 Mo
11 splayd.settings.job.disk.max_size = 1024 * 1024 * 1024 -- 1 Go
12 splayd.settings.job.disk.max_files = 1024
13 splayd.settings.job.disk.max_file_descriptors = 64
14 splayd.settings.job.network.max_send = 1024 * 1024 * 1024
15 splayd.settings.job.network.max_receive = 1024 * 1024 * 1024
16 splayd.settings.job.network.max_sockets = 64
17 splayd.settings.job.network.max_ports = 2
18 splayd.settings.job.network.start_port = 22000
19 splayd.settings.job.network.end_port = 32000
20
21 -- Information about your connection (or your limitations)
22 -- Enforce them with trickle or other tools
23 splayd.settings.network.send_speed = 1024 * 1024
24 splayd.settings.network.receive_speed = 1024 * 1024
```

Listing 3.1 – splayd's default configuration

When bootstrapping, the splayd generates a random session token. This session token is sent to the controller as part of the initialization protocol. After a disconnection, when the splayd reconnects, it will send again this token. If the token has not changed, the controller knows that the splayd was just temporarily disconnected and not restarted, so it assumes the splayd state has not changed and does not need to be reinitialized.

During the registration phase, the daemon receives a blacklist of forbidden addresses expressed as IP or DNS masks. By default, the addresses of the controllers are black-

listed so that applications cannot actively connect to them. Blacklists can be updated by the controller at runtime (e.g., when adding a new daemon or for protecting a particular machine).

The daemon also receives the address of a `log` process to connect to for logging, together with a unique identification key. SPLAY applications instantiated by the local daemon can only connect to that log process; other processes will reject any connection request.

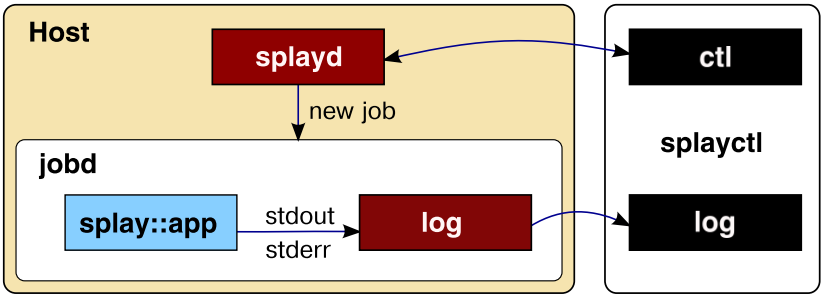


Figure 3.3 – Instance of a new job on the host and connections with the controller.

When receiving a new job, the splayd process will first create a new jobd process (see Figure 3.3) with a new Lua environment (completely isolated of the splayd Lua code for additional security) and transmit it the job's configuration and code.

The jobd process, will then launch a log process that will immediately connect with its controller alter ego. The jobd process redirects its standard output streams (stdout and stderr) to the log process. The log system will receive both the output of the job that is executed, but also every errors that could be outputted from the jobd process itself or from the SPLAY library.

Then, the jobd process cleans its environment, prepares an empty folder for the job's temporary files and sets up the Lua sandbox with the resources limitations bundled with the job that needs to be executed. Finally, the job is started.

The splayd process will monitor the state of the jobd processes and can also kill them if it receives the command from the controller.

3.4 Deployment

The deployment is one of the most important mechanism of SPLAY. It must select an appropriate set of nodes to perform a successful experiment. In this section, we will detail the internals of these mechanisms.

As seen in the previous section, each splayd has its own limits regarding the maximum resources that can be allocated for a job it will accept. For that reason, considering the resources requested, the controller will first select only splayds that can accept the job that needs to be executed. Other parameters can also be considered like the general availability of a node, its uptime or, if the job requests them, geographic criteria like using only nodes from one specific country or continent or only nodes inside a limited radius.

The controller sends a `REGISTER` message to the splayd of every selected nodes containing the application code and the resources requested. In case the identity of the nodes is not explicitly specified, the system selects a set larger than the one originally requested to account for failed or overloaded nodes.

```
1  job = {}
2  job.ref = "DF7A329B3A617E5E"
3  job.name = "Hello␣world"
4  job.description = "Print␣hello␣world"
5  job.code = "print('Hello␣world')"
6  job.debug = true
7  job.network = {}
8  job.network.max_send = 100 * 1024 * 1024
9  job.network.max_receive = 100 * 1024 * 1024
10 job.network.max_tcp_sockets = 100
11 job.network.nb_ports = 5
12 job.max_mem = 10 * 1024 * 1024
13 job.disk.max_size = 100 * 1024 * 1024
14 job.disk.max_file_descriptors = 100
15 job.disk.max_files = 100
```

Listing 3.2 – Example job's config

Listing 3.2 illustrates what is sent by the controller, job.ref is a generated unique identifier and job.code contains the applications code. Most other parameters are optional and concern resources limits.

Upon accepting the job, a splayd sends to the controller the range of ports that are available to the application. Once it receives enough replies, the controller first sends to every selected SPLAY a `LIST` message with the addresses of some or all participating nodes (depending of the job configuration) to bootstrap the application.

```
1 list = {}
2 list.ref = "DF7A329B3A617E5E"
3 list.nodes = []
4 list.nodes[1] = {ip = "193.234.12.13", port = 11090}
5 list.nodes[2] = {ip = "174.121.98.104", port = 11000}
6 list.nodes[3] = {ip = "212.34.111.48", port = 11010}
7 list.type = "head"
8 list.position = 2
```

Listing 3.3 – Job's nodes list

Listing 3.3 shows a job where each node receives the complete list of every participating nodes. We see the IP address of each node and the first port in the port range dedicated to the application. In that particular list, we see that the current node position is at position two. The list can contain all nodes or only a subset of them (in the extreme case, a single rendez-vous node). If list.type equals "random", each node will receive a different randomized subset of all nodes. Appropriate lists can greatly reduce the bootstrap time of a protocol.

The `LIST` message is then followed by the `START` message to begin execution. Supernumerary daemons that are slow to answer and active applications that must be terminated receive a `FREE` message. The `STOP` message will be used to temporarily stop a node in case of churn simulation. The state machine of a SPLAY job is as follows:

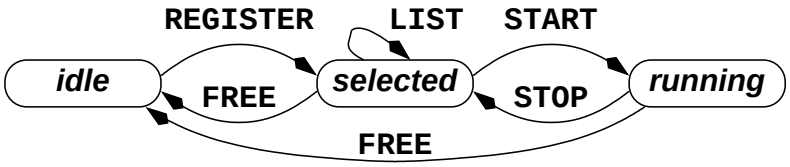


Figure 3.4 – State machine of a SPLAY job on a host.

The reason why we initially select a larger set of nodes than requested clearly appears when considering the availability of hosts on testbeds like PlanetLab, where transient failures and overloads are the norm rather than the exception. Figure 3.5 shows both the cumulative and discretized distributions of round-trip times (RTT) for a 20KB message over an already established TCP connection from the controller to PlanetLab hosts. One can observe that only 17.10% of the nodes reply within 250 milliseconds, and over 45%, need more than 1 second. Selecting a larger set of candidates allows us to choose the most responsive nodes for deploying the application.

Each `START` command is sent from the controller at the same time. Due to the network delay, each node will receive it at a slightly different time. We could implement a

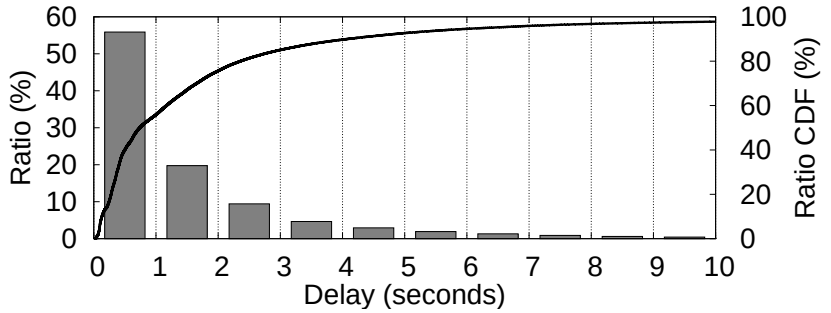


Figure 3.5 – RTT between the controller and PlanetLab hosts over pre-established TCP connections, with a 20 KB payload.

rendez-vous point in a close future to achieve a better synchronization. However, this will require every host to have their clock synchronized or implement a complete time synchronization mechanism directly inside all splayds.

3.4.1 Complex Network Configurations

The way SPLAY wraps low level network calls (to add non blocking events and sandbox layers) helps to setup complex network configurations.

A concrete example of this usage is to rewrite source and destination IPs directly inside an application. This allows us, for example, to connect a set of internal nodes in a local cluster (private IPs) with nodes located outside (public IPs).

At the network level, this configuration requires to establish a gateway. Assuming that the internal nodes have the ability to communicate directly with those outside via a classic NAT setup, to make the reverse operation possible we setup a gateway with forwarded ports to reach each listening ports of each internal nodes.

In the case where an external node wants to contact an internal node, the internal node's IP must be replaced by the gateway public IP and the port that will forward the connection to the desired internal IP and port.

In some of our experiments, the splayds had been modified to do a rewrite when the connect() function was called. If the current node was external and was trying to connect to an internal one, IP and port were rewritten to use the gateway.

We plan to add a more generic mechanism in the future, where network configuration and rewriting rules are sent directly from the controller as part as the deployment protocol.

3.5 SPLAY Applications

3.5.1 Lua

SPLAY applications are written in the Lua language [60], whose features are extended by SPLAY's libraries.

Lua is an open source scripting language created in 1993 and designed to be embedded [53] in other applications in order to extend them. It has been developed by Luiz Henrique de Figueiredo, Roberto Ierusalimschy and Waldemar Celes, members of the research group TeCGraf in the university of Rio de Janeiro in Brazil.

The choice of Lua was dictated by four majors factors.

First, Lua has unique features that allow us to simply and efficiently implement sandboxing. As mentioned earlier, sandboxing is a sound basis for execution in non-dedicated environments, where resources need to be constrained and where the hosting operating system must be shielded from possibly buggy or ill-behaved code.

Second, one of SPLAY's goals is to support large numbers of processes within a single host of the testbed. This calls for a low footprint for both the daemons and the associated libraries. This excludes languages such as Java that require several megabytes of memory just for their execution environment. The full interpreter is less than 200 kB and can be easily embedded. Applications can use libraries written in different languages (especially C/C++). This allows for low-level programming if need be. Our experiments (Section 4) highlight the lightweightness of SPLAY applications using Lua, in terms of memory footprint, load, and scalability. The small footprint of Lua results from its design that provides flexible and extensible meta-features, rather than a complete set of general-purpose facilities.

Third, SPLAY must ensure that the achieved performance is as good as the host system permits, and features offered to the distributed system designer shall not interfere with the performance of the application. The Lua language was designed from the ground up to be very efficient. According to recent benchmarks [5], Lua is among the fastest interpreted scripting languages.

Fourth, SPLAY allows deployment of applications on any hardware and any operating systems. This requires a "write-once, run everywhere" approach that calls for either an interpreted or bytecode-based language. Lua's interpreter can directly execute source code, as well as hardware-dependent (but operating system-independent) bytecode. In SPLAY, the favored way of submitting applications is in the form of source code, but bytecode programs are also supported (e.g., for intellectual property protection).

Lua supports both the imperative paradigm and the functional one. The language is also reflective, dynamically typed, and has automatic memory management with incremental garbage collection.

In the following sections, we give a short overview of the most notable features of the language.

First class functions and curryfication

Functions are first class objects. They can be nested or anonymous. They also support variable number of arguments and multiple return values.

```
1  function apply(a, f)
2    for i, e in ipairs(a) do
3      a[i] = f(e)
4    end
5    return a
6  end
7
8  a = {1, 2, 3}
9
10 -- Passing anonymous function as second parameter
11 a = apply(a, function(a) return 2 * a end)
12
13 -- => a = {2, 4, 6}
```

Listing 3.4 – Functional example

Lua support for higher-order functions permits to easily implements curryfication [1]:

```
1  function curryfy(f, v)
2    return function (...) return f(v, ...) end
3  end
4
5  f = function(e, f) print(e, f) end
6  f('hello', 'world') -- => hello world
7
8  -- curryfication of the function f
9  g = curryfy(f, 'hello')
10 g('world') -- => hello world
```

Listing 3.5 – Currification of a function

[1] Currying consists of transforming a function that takes multiple arguments into a chain of functions each with a single argument (partial application).

Closures and Lexical Scoping

A closure is a function together with a referencing environment. The referencing environment is the lexical scope at the position where the function is defined. The captured variables outside the functions are called upvalues.

```
1  -- closure created using a function which define an internal function
2  function say_hello(name)
3    local phrase = "Hello "..name
4    return function()
5      print(phrase)
6    end
7  end
8
9  say = say_hello("world")
10 say() -- => "Hello world"
11
12 -- closure created using a block
13 do
14   -- backup of the previous existing function
15   local s_h = say_hello
16   -- "upvalue" of say_hello()
17   local w = "Bobby"
18
19   function say_hello()
20     -- reusing the previous function
21     local say = s_h(w)
22     say()
23   end
24 end
25
26 -- this will not affect the variable in the closure
27 w = "world"
28
29 say_hello() -- => "Hello Bobby"
```

Listing 3.6 – Closure and lexical scoping

The SPLAY sandbox extensively use these features to save and replace existing functions by sandboxed ones.

Tables and Metatables

The only native structure of the language is the table. Lua tables can behave both as indexed arrays or hashes depending of the functions using them (they can have both behavior at the same time).

Tables can store any kind of Lua objects, from numbers to functions. Every table can have a metatable attached to it, which, with some certain keys set, can change the behavior of the table it is attached to.

```
1  -- two simple tables used as arrays
2  t1 = {1, 2, 3}
3  t2 = {4, 5, 6}
4
5  -- a metatable containing a metamethod that define the addition
6  -- operation (addition is by default not defined between tables)
7  local mt = {
8
9    __add = function(a, b)
10     local result = {}
11     for i in ipairs(a) do
12       result[i] = a[i] + b[i]
13     end
14     return result
15   end
16
17 }
18
19 -- we affect the metatable to t1
20 setmetatable(t1 ,mt)
21
22 -- now we can add the 2 tables
23 t1 = t1 + t2
24
25 -- => t1 = {5, 7, 9}
```

Listing 3.7 – Simple metatable example

Lua does not directly support OOP (Object Oriented Programming) using traditional class constructs, but instead, using the power of metatables and metamethods, it can provide a very similar behavior (including inheritance mechanisms), using prototype objects. This approach is called Prototype-based OOP.

Coroutines

Lua also supports cooperative multitasking by means of coroutines:

```
1
2  function hw()
3    print('hello')
4    coroutine.yield() -- leave the coroutine, store the context
5    print('world')
6  end
7
8  co = coroutine.create(hw)
9  coroutine.resume(co) -- => hello
10 coroutine.resume(co) -- => world
```

Listing 3.8 – Coroutines

Using coroutines coupled with a scheduler, we have built a cooperative multitasking system, which we describe next.

3.5.2 The Scheduler

Lua complies with the ANSI C standard that has no mechanism for managing multiple threads of execution. The threads and the synchronization objects can be provided by the underlying operating system, but system threads coordination and avoiding shared data structures corruption is a tricky task.

Fortunately, Lua provides an alternative solution that alleviates this problem by providing single-threaded coroutines. Coroutines permit us to have multiple points of execution by switching from one to another in a cooperative process (the current coroutine pa control to another coroutine).

However, manually managing coroutines is a tedious process that greatly increases the application complexity and is also error prone. To hide this complexity and to create a useful multi-tasking programming environment, we have designed our own coroutine scheduler directly written in Lua.

In a cooperative scheduler, every coroutine should, at some point, decides to interrupt itself and to call the scheduler. This complexity is hidden inside new functions that replace (wrap) the original functions doing blocking IO.

The scheduler will then have a set of resources to monitor and a set of coroutines waiting on them to be available for reading or writing. Another problem to solve is to avoid an infinite loop that checks the status of the resources all the time. This is solved using the select() system call.

This corresponds to an implementation of the Reactor pattern [43] where the Synchronous Event Demultiplexer is played by the select() function, the Dispatcher is our scheduler and the Request Handler is the coroutine that uses a socket (the resource).

To propose a better "threaded" programming environment, we have also added support for our coroutines to sleep or to wait for a specific event.

For the user point of view, all network functions will be blocking (as generally expected in threaded applications), but behind the scene, each of them calls the scheduler that will queue the current coroutine and add its socket to the select list (see Figure 3.6).

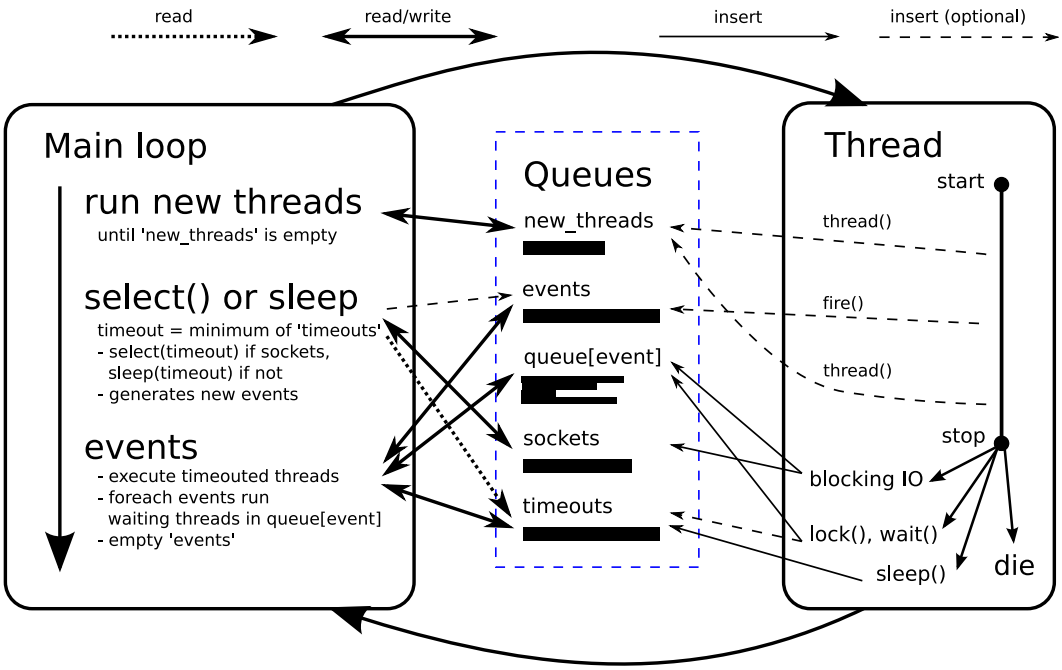


Figure 3.6 – The scheduler

The use of select() in the scheduler is fundamental to avoid "active" waiting (consuming CPU), by providing a list of file descriptors (sockets are file descriptors) to the select() system call, permitting the application to sleep until the next event or the next timeout.

The SPLAY scheduler manages several queues for coroutines waiting on sockets, on internal events or sleeping for a given period. The scheduler is smart enough to call the select() function with a timeout corresponding to the nearest next timeout of the timeout queue. This method permits us to completely avoid active waiting.

In some applications, there could be processes with 'long' computations ($>$ 1ms), not doing any network I/O. In that situation, it is recommended to insert manual scheduling calls inside it to keep the responsiveness of the application.

Compared to a language that provides system threads, a SPLAY application has the disadvantage to only use at most one physical CPU core (when system threads can be dispatched on multiple cores) but it also benefits of the advantage to be very lightweight while using the very efficient non blocking I/O mechanism transparently for the developer.

3.5.3 The Libraries

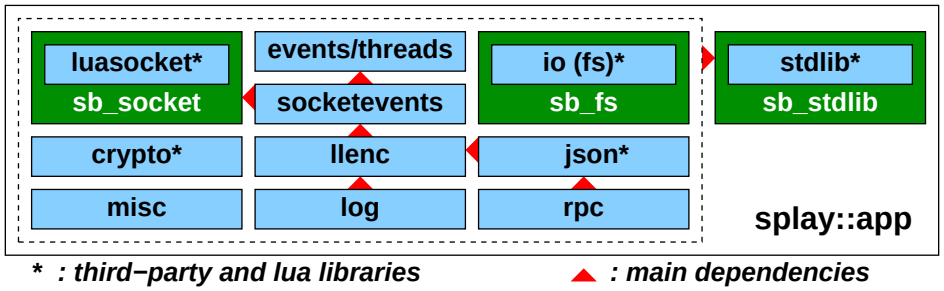


Figure 3.7 – Overview of the main SPLAY libraries.

SPLAY includes an extensible set of shared libraries (see Figure 3.7) tailored for the development of distributed applications and overlays. These libraries are meant to be also used outside of the deployment system, when developing the application. We describe the major components of these libraries.

3.5.3.1 Networking

The `luasocket` library provides standard low level networking facilities [1]. We have wrapped it into a restricted socket library, `sb_socket`, which includes a security layer that can be controlled by the local administrator (the person who has instantiated the local daemon process) and further restricted remotely by the controller. Restrictions are specified declaratively in the configuration file (see listing 3.1, network part) by the local user that starts the daemon, or at the controller via the command-line and Web-based APIs.

[1] Available from http://www.cs.princeton.edu/~diego/professional/luasocket/.

This secure layer allows us to limit:

1. The total number of opened sockets.
2. The number of listening ports (a range is specified).
3. The maximum number of bytes sent and received.
4. Hosts to which applications can not connect (blacklist expressed as IP or DNS masks).

(1) and (2) directly concern local system resources while (3) and (4) constrain communications with the outside world.

Wrapping network functions was not straightforward. Socket objects provided by `luasocket` change their states: a master socket transforms into a server socket when binded; a (TCP) server sockets returns client sockets that should also be securely wrapped. Our security layer permits us to keep complete compatibility with `luasocket` while providing additional stats() and limits() functions to applications who can access resource usage while running and check the (hard) limits.

However, the layer does not apply any kind of traffic shaping. This can be obtained more efficiently using system specific tools. On Unix, different solutions are available like QoS (that require administrator privileges) or tools like 'trickle', a portable lightweight userspace bandwidth shaper.

We also have implemented higher-level abstractions for simplifying communication between remote processes. Our API supports message passing over TCP and UDP, as well as access to remote function and variables using RPCs (more on this in section 3.6). Calling a remote function is almost as simple as calling a local one (see the code in the next section). All arguments and return values are transparently serialized. Communication errors are reported using a second return values, as allowed by Lua.

Finally, communication libraries can be instructed to drop a given proportion of the packets (specified upon deployment): this can be used to simulate lossy links and study their impact on an application.

3.5.3.2 Virtual Filesystem

Overlays and distributed applications often need to use the local file system. For instance, when instantiating the BitTorrent protocol to replicate a large file on a set of

nodes, temporary data must be written to disk as chunks are being received. Following our goal to not impact the hosting operating system, we need to ensure that a SPLAY application cannot access or overwrite any data on the host file system.

To this end, SPLAY includes a library, `sb_fs`, that wraps the standard `io` library and provides restricted access to the file system in an OS-independent fashion.

The restrictions include:

- Total file system usage.
- Number of files.
- Number of opened files (file descriptors).

We follow the standard Lua I/O API to provide file functions. First we wrap the open() function, then we return a wrapped handle that will do some additional things: store the path + filename (user view) in a memory table and do a mapping between the filename and the real file on the file system by hashing the filename.

Every successive access to the file is redirected in a single application-specific folder on the real file system under a name that results from the SHA hashing of the original filename and path.

An application can only see what was directly created by itself. SPLAY completely denies real file system access or avoid path related vulnerabilities. The wrapped handle will manage additional restriction like disk space limit, max files (because there is a one to one mapping with the file system) and max file descriptors.

As for restricted sockets, two additional functions stats() and limits() are provided to the application to monitor its current usage and adapt its behavior.

3.5.3.3 Events, Threads and Locks

SPLAY proposes a threading model based on Lua's coroutines combined with event-based programming. Unlike preemptive threads, coroutines yield the processor to each other (cooperative multitasking). This happens at special points in base libraries, typically when performing an operation that may block (e.g., disk or network I/O). This is typically transparent to the application developer. Although a *single* SPLAY application will not benefit from a multicore processor, coroutines are preferable to system-level threads for two reasons: their portability and their recognized efficiency

(low latency and high throughput) for programs that use many network connections (using either non-blocking or RPC-based programming), which is typical of distributed systems programming. Moreover, using a single process (at the operating system level) has a lower footprint, especially from a sandboxing perspective, and allows deploying more applications on each `splayd`.

Shared data accesses are also safer with coroutines, as race conditions can only occur if the current thread yields the processor. This requires, however, a good understanding of the behavior of the application (we illustrate a common pitfall in Section 3.7). SPLAY provides a lock library to protect shared data from concurrent accesses by multiple coroutines.

We have also developed an event library, `events`, that controls the main execution loop of the application, the scheduler, the communication between coroutines, timeouts, as well as event generation, waiting, and reception. To integrate with the event library, we have wrapped the socket library to produce a non-blocking, coroutine-aware version `sb_socket`. All these layers are transparent to the SPLAY developer who only sees a restricted, non-blocking socket library.

3.5.3.4 Logging

An important objective of SPLAY is to be able to quickly prototype and experiment with distributed algorithms. To that end, one must be able to easily debug and collect statistics about the SPLAY application at runtime. The `log` library allows the developer to print information either locally (screen, file) or, more interestingly, send it over the network to a log collector managed by the controller. If need be, the amount of data sent to the log collector can be restricted by a `splayd`, as instructed by the controller. As with most log libraries, facilities are provided to manage different log levels and dynamically enable or disable logging.

3.5.3.5 Other libraries

SPLAY provides a few other libraries with facilities useful for developing distributed systems and applications. The `llenc` and `json` libraries support automatic and efficient serialization of data to be sent to remote nodes over the network. We developed the first one, `llenc`, to simplify message passing over stream-oriented protocols (e.g., TCP). The library automatically performs message demarcation, computing buffer sizes and waiting for all packets of a message before delivery. It uses the `json` library to automate encoding of any type of data structures using a compact and standardized

data-interchange format. The `crypto` library includes cryptographic functions for data encryption and decryption, secure hashing, signatures, etc. The `misc` library provides common containers, functions for format conversion, bit manipulation, high-precision timers and distributed synchronization.

The memory footprint of these libraries is remarkably small. The base size of a SPLAY application is less than 600 kB with all the abovementioned libraries loaded. It is easy for administrators to deploy additional third-party software with the daemons, in the form of libraries. Lua has been designed to seamlessly interact with C/C++, and other languages that bind to C can be used as well. For instance, we successfully linked some SPLAY application code with a third-party video transcoding library in C, for experimenting with adaptive video multicast. Obviously, the administrator is responsible for providing sandboxing in these libraries if required.

3.6 Remote Procedure Call

The primary reason to develop RPCs is to provide a very easy, natural and elegant way to call a remote function and get its return values. The RPC layer greatly reduces the amount of code when designing applications and avoids the need to implement a dedicated transmission protocol.

When calling a local function, the values received are exactly those returned by the function (not considering exceptions here). The time between the function call and the return corresponds approximately to the execution time of the function.

The first problem when using call over the network is the delay. Depending of the size of the parameters, the size of return values, the connection delay and the network speed, the total calling time of a function can greatly varies. As a rule of thumb, one should only use RPCs for functions that do not exchange many data and use TCP streams otherwise.

The second problem is to distinguish between function errors and network errors. We must provide enough information to the developers for the application to be able to deal with network failures if needed.

SPLAY RPCs provide mechanisms to handle errors, but the programmer must be aware of them when designing his application and consider, at least, the additional network delays.

In Splay, three versions of RPCs have been implemented: using TCP connections, using UDP connections and using a pool of TCP connections. Transparently switching between RPC implementations has permitted us to detect some system-related problems (for example, the TCP implementation use more file descriptors and resources than the UDP one), ensuring that each implementation is functional and under different conditions.

The choice between one implementation or another will depend of various parameters like the transfer size of functions, if multiple asynchronous calls on a same node are needed or if the total amount of memory must be kept very low. We details some design characteristics of these implementations in the next sections.

3.6.1 Implementations

Splay RPCs are built on the Splay network stack (socket events). Although the implementation between TCP and UDP is completely different, we tried to keep the usage compatible as much as possible for the developer.

Each TCP connection needs one dedicated socket on both end. Sockets are a resource provided by the operating system. Each socket generally uses 4KB of operating system memory. For that reason, sockets are valuable and limiting their usage can be very important depending of the experiment.

Both TCP and UDP implementations use bencoding [6] to encode data. This encoding was originally used by BitTorrent. It permits us to transfer data structures and adds a very low additional transmission overhead.

Each participating peer has a local TCP (respectively UDP) server that listen for incoming RPC calls. Applications can use both TCP and UDP RPCs at the same time.

3.6.1.1 TCP

TCP is a data transfer protocol that provides reliable, ordered and error-checked delivery of a stream of bytes between two connected applications. It requires handshaking to set up end-to-end communications (three packets exchange).

Our first TCP implementation uses exactly one TCP connection for each RPC call. Once a call is completed, the TCP connections are immediately closed.

This implementation has the advantage to be asynchronous and to free the resources (TCP connections) as soon as possible. However, in situations where many connections are done with the same peer, using a single asynchronous TCP connection could avoid the cost of establishing one TCP connection for every call.

3.6.1.2 UDP

Unlike TCP which is stream-oriented, UDP has no session semantics. It does not use handshaking and provides no transmission guarantees. An internal checksum, nonetheless, is used to verify the integrity of individual datagrams.

The theoretical maximum size of an UDP datagram is 64KB but, practically the send/receive buffer size of the operating system is less than this maximum. A maximum size of 8KB for our UDP RPC datagram seems to be a safe choice with today's operating systems. One should also note that the smaller the datagram is, the smaller the delay (a bigger datagram will possibly be split by the underlying IP layer depending on the MTU value (maximum transmission unit or the various links)).

This implementation has been done for two reasons. First, to avoid resources usage of TCP sockets (each peer requires only one UDP socket). Secondly, to benefit from the lower delays that can be achieved using UDP (there is no need to establish a connection as with TCP).

Considering that most function calls do not require exchanging more than 8KB of data, UDP is an excellent choice if we can add some reliability guarantees. Splitting data between UDP datagrams would, more or less, corresponds to reimplement TCP over UDP, loosing simplicity, efficiency and effectively reinventing the wheel.

We call our RPC implementation over UDP: 'uRPC'.

In our implementation, each RPC message receives an unique identifier which is the concatenation of a unique host identifier and a local counter. This identifier is essential to detect message duplication and act accordingly. Each peer has a local cache of all messages received, indexed by their identifiers.

A standard uRPC exchange consists of three messages. The first message, sent from the client to the server, contains the function to call and its parameters. Once received, the server will execute the corresponding local function, cache the return values of the function and reply to the client. Finally, the client sends a 'free' message that will be used by the server to remove the entry in its local cache.

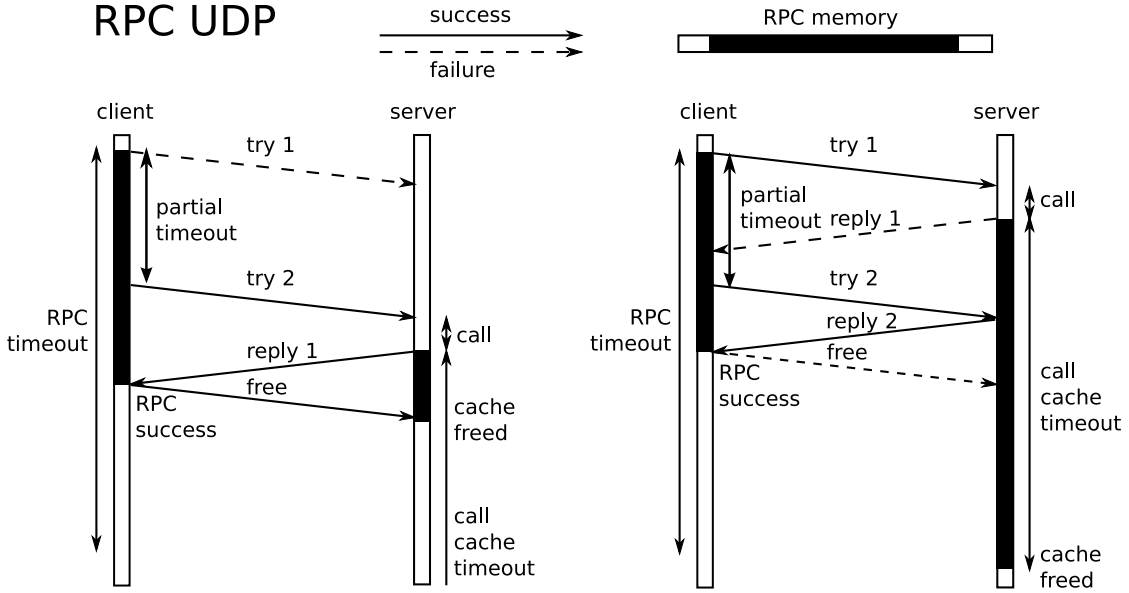


Figure 3.8 – uRPC failures and error correction

Considering the guarantees offered by UDP, each of these messages can potentially be lost. In order to avoid semantic problems with the RPC, we have set up some additional mechanisms.

In our first example (left side of figure 3.8), the message 'try 1' is lost and the client retries with 'try 2' after a timeout. When the server receives this message, it replies with 'reply 1', and finally, the client sends the 'free' message.

In the second example (right side of figure 3.8), the server has received 'try 1' but 'reply 1' has been lost. In that case, as for the first example, the client will, after a timeout, re-emit the message ('try 2'). The server receiving that message for the second time, will detect an already existing local cache entry for this message and will not call again the RPC function (calling again the same function can have undesirable results). Instead, it will use the cached data to directly reply with 'reply 2'. The client receives it and sends back a 'free' message to the server. In this case, this message will be lost, and the server not able to free the cache immediately. However, each cache entry has a global timeout that will eventually free it.

The role of the 'free' message is purely a memory optimization. It permits us to free the server cache as soon as possible and prevents it from retaining too much unnecessary data (each cache entry can contain up to 8KB of data).

3.6.1.3 TCP pool

The TCP pool implementation is a way to keep the resources below a certain level by limiting the number of opened connections and using them to transmit multiple RPCs.

The TCP pool is specially useful when peers are connected with a limited number of other peers and need to exchange messages for a significant amount of time. As only one TCP connection (one for each direction) is opened between two peers, the calls and the replies are no more asynchronous but serialized. This can be a major drawback, specially if the size of the data exchanged by some RPC calls is substantial.

The maximum number of connections is limited by the size of the TCP pool. Once the limit is reached, one connection has to be closed before being able to create a new one (the least recently used).

Actually in our implementation of the TCP pool, the connections are not used symmetrically: only the peer having opened the connection can send a RPC through it. The reason is to simplify the mode of operation without having to set up a closing protocol. The initiator peer can decide itself when to close the connection once the current RPC call has completed.

One of the remaining problems is that the number of sockets used by the current peer also depends of the number of connections made to that peer by all other peers (a popular peer can receive many connections). Considering that is not acceptable to refuse a new connection, once the number of sockets used is too important, we should inform the connecting peer that we just want an ephemeral TCP connection (similar to the simple TCP implementation) containing only one RPC call.

This RPC implementation is still at an early stage and some improvements are still needed such as using the TCP connection between two peers symmetrically and thus further reducing the resource requirements.

3.6.2 Usage

3.6.2.1 Error detection

Different high level functions are provided to the developer for convenience:

call() This function does not perform any error handling. It directly return the result of the remote function. If there is any network problem, it replies in the Lua

common error format (nil, 'error'). It is only useful for rapid prototyping (local machine or internal network) or if the remotely called function never replies with 'nil': by deduction one will know it was a network error. If the remote function can also sometimes reply (nil, 'error'), one will not be able to distinguish the errors.

acall() This function provides all the needed information to detect a network problem. The first returned value indicates if the call, at the network level, was successful or not. The second value reports the error code (if any) or an array containing the function's returned values.

ecall() This function returns the same values as 'call()' but throws an exception (Lua error) if a network problem occurs.

3.6.2.2 Timeouts

When using RPCs, one can associate a timeout to each call. This timeout include the whole time to complete the RPC: network delays and the time to execute the remote function. The timeout value can often be difficult to choose:

- We might have no precise idea of how much time the remote function will need to complete (the remote function can also call other RPCs or do some network operations before returning).
- The remote function can return a big amount of data that will take time for transfer.

The first problem is very common in P2P overlays because finding the right node is traditionally a recursive operation. That means the number of recursive calls can vary significantly. In that situation, the developer can still use RPCs, but can also use a callback mechanism.

Technically, RPC implementations based on TCP can transfers an unlimited amount of data. The difficulty is to increase the timeout values (or not using them) to have sufficient time for transfers. However, transferring a large amount of data using RPCs is probably a bad idea and traditional TCP connections are more appropriate.

3.6.2.3 Callbacks

Callbacks are not directly a SPLAY mechanism, but a way to use RPCs. The called function will always reply immediately without response (the RPC call will be finished immediately). However, the function will launch a new thread to call an RPC on the next peer in the case of chained RPC calls between peers. The last peer of the recursion should directly send the reply to the origin peer (information from the originator peer are transmitted through the recursive calls). This scheme has also the advantage of avoiding transferring the data through all the peers in the recursion, thus saving network traffic and lowering the delay.

```
1  require "splay.base"
2  rpc = require "splay.rpc"
3
4  me = {ip = "127.0.0.1", port = 2000}
5
6  function callback(node, count)
7    print("result after "..count.." nodes")
8    events.fire("node", node)
9  end
10
11 function recursive_find(m)
12   -- Creating a new thread, the function will return immediately
13   events.thread(function()
14     if math.random(10) == 1 then
15       -- i am the destination
16       rpc.call(m.origin, {'callback', me, m.count})
17     else
18       m.count = m.count + 1
19       local next = me -- no budget for more nodes
20       rpc.call(next, {'recursive_find', m})
21     end
22   end)
23 end
24
25 function find()
26   local m = {origin = me, count = 0}
27   rpc.call(me, {'recursive_find', m})
28   return events.wait("node")
29 end
30
31 events.run(function()
32   rpc.server(me)
33   local dest = find()
34   print(dest.ip, dest.port)
35   os.exit()
36 end)
```

Listing 3.9 – Example of a recursive find with a callback.

Callbacks are especially advised for functions with recursive behavior. They permit to transform a potentially slow blocking mechanism in a non-blocking asynchronous one. As callback programming may not be very natural, another way to use them is by calling back a function on the originator peer that will transform the callback into a local event (see function 'find()' in Listing 3.9).

3.7 Developing Applications with SPLAY

This section illustrates the development of an application for SPLAY. We use the well-known Chord overlay [121] for its familiarity to the community. As we will see, the specification of this overlay is remarkably concise and close to the pseudo-code found in the original paper. We have successfully deployed this implementation on a ModelNet cluster and PlanetLab; results are presented in Section 4.3. The goal here is to provide the reader with a complete chain of development, deployment, and monitoring of a well-known distributed application. Note that local testing and debugging is generally done outside of the deployment framework (but still, using SPLAY libraries).

Chord is a distributed hash table (DHT) that maps keys to nodes in a peer-to-peer infrastructure. Any node can use the DHT substrate to determine the current live node that is responsible for a given key. When joining the network, a node receives a unique identifier (typically by hashing its IP address and port number) that determines its position in the identifier space. Nodes are organized in a ring according to their identifiers, and every node is responsible for the keys that fall between itself (inclusive) and its predecessor (exclusive). In addition to keeping track of their successors and predecessors on the ring, each node maintains a "finger" table whose entries point to nodes at an exponentially increasing distance from the current node's position. More precisely, the i^{th} entry of a node with identifier n designates the live node responsible for key $n + 2^i$. Note that the successor is effectively the first entry in the finger table.

Listing 3.10 shows the code for the construction and maintenance of the Chord overlay. For clarity, we only show here the basic algorithm that was proposed in [121] (the reader can appreciate the similarity between this code and Figure 6 of the referenced paper).

```
1  function join(n0)                                          -- n0: some node in the ring
2    predecessor = nil
3    finger[1] = call(n0, {'find_successor', n.id})
4    call(finger[1], {'notify', n})
5  end
6  function stabilize()                                       -- periodically verify n's successor
7    local x = call(finger[1], 'predecessor')
8    if x and between(x.id, n.id, finger[1].id, false, false) then
9      finger[1] = x                                          -- new successor
10   end
11   call(finger[1], {'notify', n})
12 end
13 function notify(n0)                                        -- n0 thinks it might be our predecessor
14   if not predecessor or between(n0.id, predecessor.id, n.id, false, false) then
15     predecessor = n0                                       -- new predecessor
16   end
17 end
18 function fix_fingers()                                     -- refresh fingers
19   refresh = (refresh % m) + 1                              -- 1 ≤ refresh ≤ m
20   finger[refresh] = find_successor((n.id + 2^(refresh - 1)) % 2^m)
21 end
22 function check_predecessor()                               -- checks if predecessor has failed
23   if predecessor and not ping(predecessor) then
24     predecessor = nil
25   end
26 end
```

Listing 3.10 – SPLAY code for Chord overlay (stabilization).

Function `join()` allows a node to join the Chord ring. Only its successor is set: its predecessor and successor's predecessor will be updated as part of the stabilization process. Function `stabilize()` periodically verifies that a node is its own successor's predecessor and notifies the successor. SPLAY base library's `between` call determines the inclusion of a value in a range, on a ring. Function `notify()` tells a node that its predecessor might be incorrect. Function `fix_fingers()` iteratively refreshes fingers. Finally, function `check_predecessor()` periodically checks if a node's predecessor has failed.

These functions are identical in their behavior and very similar in their form to those published in [121]. Yet, they correspond to executable code that can be readily deployed. The implementation of Chord illustrates a subtle problem that occurs frequently when developing distributed applications from a high-level pseudo-code description: the reception of multiple messages may trigger concurrent operations that perform conflicting modifications on the state of the node. SPLAY's coroutine model alleviates this problem in some, but not all, situations. During the blocking call to `ping()` on line 23 of Listing 3.10, a remote call to `notify()` can update the predeces-

sor, which may be erased on line 24 until the next remote call to `notify()`. This is not a major issue as it may only delay stabilization, not break consistency. It can be avoided by adding an extra check after the ping or, more generally, by using the locks provided by the SPLAY standard libraries (not shown here).

```
27 function find_successor(id)                          -- ask node to find id's successor
28   if between(id, n.id, finger[1].id, false, true)      -- inclusive for second bound
29     return finger[1]
30   end
31   local n0 = closest_preceding_node(id)
32   return call(n0, {'find_successor', id})
33 end
34 function closest_preceding_node(id)                    -- finger preceding id
35   for i = m, 1, -1 do
36     if finger[i] and between(finger[i].id, n.id, id, false, false) then
37       return finger[i]
38     end
39   end
40   return n
41 end
```

Listing 3.11 – SPLAY code for Chord overlay (lookup).

Listing 3.11 shows the code for Chord lookup. Function `find_successor()` looks for the successor of a given identifier, while function `closest_preceding_node()` returns the highest predecessor of a given identifier found in the finger table. Again, one can appreciate the similarity with the original pseudo-code.

```
1  require "splay.base"                          -- events, misc, socket (core libraries)
2  rpc = require "splay.rpc"                     -- rpc (optional library)
3  between, call, ping = misc.between_c, rpc.call, rpc.ping   -- aliases
4  timeout = 5                                   -- stabilization frequency
5  m = 24                                        -- 2^m nodes and key with identifiers of length m
6  n = job.me                                    -- our node {ip, port, id}
7  n.id = math.random(1, 2^m)                    -- random position on ring
8  predecessor = nil                             -- previous node on ring {id, ip, port}
9  finger = {[1] = n}                            -- finger table with m entries
10 refresh = 0                                   -- next finger to refresh
11 n0 = job.nodes(1)                             -- first peer is rendez-vous node
12 rpc.server(n.port)                            -- start rpc server
13 events.thread(function() join(n0) end)        -- join chord ring
14 events.periodic(stabilize, timeout)           -- periodically check successor, ...
15 events.periodic(check_predecessor, timeout)   -- predecessor, ...
16 events.periodic(fix_fingers, timeout)         -- and fingers
17 events.loop()                                 -- execute main loop
```

Listing 3.12 – SPLAY code for Chord overlay (initialization).

This almost completes our minimal Chord implementation, with the exception of the

initialization code shown in Listing 3.12. One can specifically note the registration of periodic stabilization tasks and the invocation of the main event loop.

While this code is quite classical in its form, the remarkable features are the conciseness of the implementation [1] , the closeness to pseudo-code, and the ease with which one can communicate with other nodes of the system by RPC. Of course, most of the complexity is hidden inside the SPLAY infrastructure.

The presented implementation is not fault-tolerant. Although the goal of this section is not to present the design of a fault-tolerant Chord, we briefly elaborate below on some steps needed to make Chord robust enough for running on error-prone platforms such as PlanetLab. The first step is to take into account the absence of a reply to an RPC. Consider the call to `predecessor` in method `stabilize()`. One simply needs to replace this call by the code of Figure 3.13.

```
1 function stabilize()                          -- rpc.a_call() returns both status and results
2   local ok, x = rpc.a_call(finger[1], 'predecessor', 60)   -- RPC, 1m timeout
3   if not ok then
4     suspect(finger[1])                        -- will prune the node out of local routing tables
5   else
6     (...)
```

Listing 3.13 – Fault-tolerant RPC call

We omit the code of function `suspect()` for brevity. Depending on the reliability of the links, this function prunes the suspected node after a configurable number of missed replies. One can tune the RPC timeout according to the target platform (here, 1 minute instead of the standard 2 minutes), or use an adaptive strategy (e.g., exponentially increasing timeouts). Finally, as suggested by [121] and similarly to the leafset structure used in Pastry [110], we replace the single successor and predecessor by a list of 4 peers in each direction on the ring.

Our Chord implementation without fault-tolerance is only 58 lines long, which represents an increase of 18% over the pseudo-code from the original paper (which does not contain initialization code, while our code does). Our fault-tolerant version is only 100 lines long, i.e., 73% more than the base implementation (29% for fault tolerance, and 44% for the leafset-like structure). We detail the procedure for deployment and the results obtained with both versions on a ModelNet cluster and on PlanetLab, respectively, in Section 4.3.

[1] As explained in [97], short programs are cheaper to build, to deploy, and to maintain.

3.8 Churn Management

The churn (contraction of the words *change* and *turn*) describes the arrival/departure behavior of nodes taking part of a network. In P2P systems, where each node has an active role in the operating of the system, it is very interesting to be able to emulate and reproduce how applications behave under churn.

In order to fully understand the behavior and robustness of a distributed protocol, it is necessary to evaluate it under different churn conditions. Theses conditions can range from rare but unpredictable hardware failures, to frequent application-level disconnections usually found in user-driven peer-to-peer systems, or even to massive failures scenarios. It is also important to allow comparison of competing algorithms under the very same churn scenarios. Relying on the natural, non-reproducible churn of testbeds such as PlanetLab often proves to be insufficient.

There exist several characterizations of churn that can be leveraged to reproduce realistic conditions for the protocol under test. First, synthetic description issued from analytical studies [98] can be used to generate churn scenarios and replay them in the system. Second, several traces of the dynamics of real networks have been made publicly available by the community (e.g., see the repository at [7]); they cover a wide range of applications such as highly churned file-sharing system [23] or high performance computing clusters [118].

SPLAY incorporates a component, `churn` (see Figure 3.2), dedicated to churn management. This component can send instructions to the daemons for stopping and starting processes on-the-fly. Churn can be specified as a trace, in a format similar to that used by [7], or as a synthetic description written in a simple script language. The trace indicates explicitly when each node enters or leaves the system while the script allows users to express phases of the application's lifetime, such as a steady increase or decrease of the number of peers over a given time duration, periods with continuous churn, massive failures, join flash crowds, etc.

Section 4.6 presents typical uses of the churn management mechanism in the evaluation of a large scale distributed system. It is noteworthy that the churn management system relieves the need for fault injection systems such as Loki [37]. Another typical use of the churn management system is for long-running applications, e.g., a DHT that serves as a substrate for some other distributed application under test and needs to stay available for the whole duration of the experiments. In such a scenario, one can ask the churn manager to maintain a fixed-size population of nodes and to automatically bootstrap new ones as faults occur in the testbed.

3.8.1 Synthetic Language

To describe the churn in a useful and understandable way, we have developed a dedicated language be can then compiled into a trace.

Our churn synthetic language contains a set of lines, each describing the evolution of the set of peers during some period or at a particular date. Each action is composed of a timing, the action type and parameters, and additional churn actions.

```
from 5 minutes to 30 minutes add 300 churn 10% per 5 minutes
\__________(a)___________/ \_(b)_/ \__(c)__/ \____(d)____/
```

(a) timing

(b) action

(c) additional churn (or 'noise')

(d) optional churn interval (by default it uses the timing interval)

Timings can be either an instant (spontaneous action, e.g., add 300 peers 1 hour after the beginning of the experiment) or a period (e.g., remove 100 peers gradually between times 2 hours and 5 hours). Quantities are either positive integers representing a fixed number of peers or a percentage of the active peers population at the instant of the action.

Different actions are possible depending of the timing type. The mosts common are 'add', 'remove' and 'const' (only for periods). If a period parameter is given, the effect of these actions is evenly distributed on it. All three actions support additional churn parameters for constant churn during a period.

The total number of peers to churn in a period is equal to the churn percentage applied to the average number of peers in the period (without churn). Technically speaking, the period (and the percentage to churn) is split in smaller ones, and the number of replacements computed for each of these smaller periods.

The churn parameter expresses how many peers will be replaced during the period, it does not affect the overall size of the peer population. When a new peer must be added, the churn system can select a previous peer switched off or a completely new one. The reuse of already existing peers is controlled by an additional parameters called the replacement ratio.

The limitations of this language is that each command acts randomly on the whole (sub)set of peers. It is impossible to set a "group" behavior for a given set of peers and a global behavior for the others (e.g., representing a small set of peers acting as super nodes with a very small churn and many highly 'churned' peers is impossible). However, compiling the churn language generates a 'trace' (see next section), that can be modified to obtain the desired behavior.

```
1  at 30s join 10
2  from 5m to 10m inc 10
3  from 10m to 15m const churn 50%
4  at 15m leave 50%
5  from 15m to 20m inc 10 churn
       150%
6  at 20m stop
```

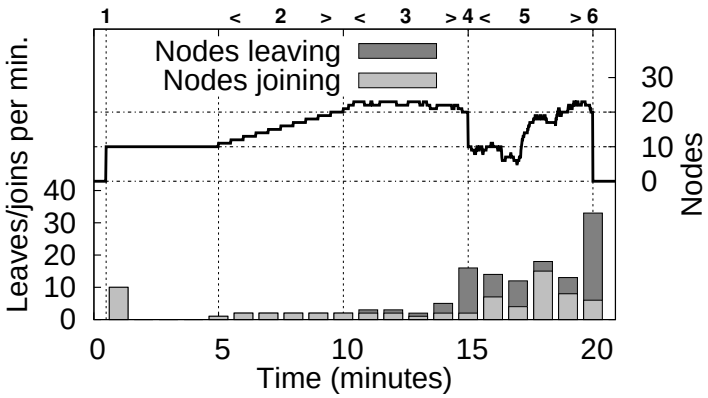


Figure 3.9 – Example of a synthetic churn description.

An example script is shown in Figure 3.9 together with a representation of the evolution of the node population and the number of arrivals and departures during each one-minute period: an initial set of nodes joins after 30 seconds, then the system stabilizes before a regular increase, a period with a constant population but a churn that sees half of the nodes leave and an equal number join, a massive failure of half of the nodes, another increase under high churn, and finally the departure of all the nodes.

3.8.2 Controller's Job Trace

The controller language for churn is very simple. It consists of a textual representation where each line contains the status of one particular peer. Each number in the line indicates when (in seconds from the start of the experiment) the peer needs to change its status: if the peer is running it will be shut down, if not running it will be started. This description is called the 'trace' of the job. Peers that have an even number of times in their line remain active at the end of the trace.

Example of job's trace with 6 peers:

```
0 600
0 300 600
0 300 600
240 1200
240 1200
600
```

In this example, the three first peers are started from the beginning (time = 0). The first one stops permanently after 10 minutes. The second and third peers quit at 5 minutes and come back at 10 minutes. Two additional peers joins after 4 minutes and stop after 12 minutes. Finally, a new peer comes after 10 minutes and stay forever.

When a peer starts (either first start or a subsequent start), it receives a list (full or partial, depending of the job's configuration) built using the peers *currently* online (the peers that start at the same time are also considered online).

The controller will register the job on as many splayds as there are rows in the trace. The `churn` process parses the trace and converts it in actions (START, STOP and LIST) that will be sent to the corresponding splayd at the appropriated time.

The job lasts until the max time limit is reached or until the trace is finished and every peer is stopped.

For long running jobs on instable testbeds (PlanetLab for example), the controller has an additional feature that will try to minimize the impact of external churn on the splayds: As for normal jobs, the controller registers the job on more peers than needed. But, unlike traditional jobs, the supernumerary splayds are not freed but kept as spare splayds. In case a splayd disappear, its trace will immediately be reallocated to another one, thus, limiting the impact of the undesired churn on the population of peers.

3.9 Sandboxing

Sandboxing is a security mechanism that isolates a running application. This mechanism restricts the scope of the application to a well defined set of allowed resources. Sandboxing protects against malicious usage (untrusted code) and programming errors. A sandboxed process must not be able to access resources outside its allowed scope.

In addition to providing security, a sandbox should also provide a stable, consistent environment where each execution of the program will be guaranteed to operate in similar conditions.

Sandboxing quickly emerged as a fundamental feature for SPLAY. Being able to run untrusted applications while giving high security guarantees to the host is essential to access a broader set of systems and hardware owned by different users and organizations.

In this section, we will give a more in depth understanding of why we made the choice to use language-based sandboxing and specifically Lua. For this purpose, we will first describe our needs, and then evaluate them against existing solutions.

To fit our requirements for SPLAY, the following characteristics were essentials:

Transparency: The same code runs inside and outside the sandbox without modifications.

Granularity: Different restrictions can be applied for each application.

Alter function calls: One can modify the behavior (parameters and return values) of sandbox functions depending of the context.

Low overhead: Overhead (CPU, memory and disk) should be very low in order to support many nodes on a single computer.

User level security: No administrator rights are needed to execute the daemon.

These extra characteristics were also important for us:

Information: The application can optionally get information from the sandbox to adapt its usage depending of the amount of remaining resources.

Portability: The application can work on many different architectures.

Ecosystem: Ability to reuse existing code and libraries in the application.

The SPLAY sandbox must be able to manage following three resources:

- Memory usage;
- Scratch space on the file system;
- Network usage.

A SPLAY application is typically a networked application, there is no need to access devices like screen, sound card, keyboard, etc.

Network access must specially be finely tunable. We could for example desire to limit an experiment to a specific list of IPs and ports and completely eliminate all risks of attacks to external targets.

CPU time will generally not be directly controlled by the sandbox itself. This limitation is generally enforced by a system specific CPU priority given to the process.

3.9.1 Evaluation of Existing Sandboxing Solutions

To explain our choices, we analyze in more details how it is possible to secure our applications using various existing technologies:

- Language-level VMs (Virtual Machines);
- Unix-like [1] security mechanisms;
- Unix-like isolation;
- Library interposers;
- Emulation and virtualization (full system VMs);
- Language level.

[1] Systems that behave similar to a Unix system, without necessarily conforming to all the associated specifications, with additional specific characteristics. Mac OS X, Linux, FreeBSD or Solaris are some well known examples.

3.9.1.1 Language-level Virtual Machines

Language-level VMs executes a specific virtual machine code commonly called "bytecode". The bytecode is an intermediate language generated from a programming language (in some cases, multiple source languages can be used) and compiled for a particular virtual machine. Well known examples of virtual machines are Sun Java JVM, Microsoft .NET CLR and Adobe Flash.

Sandboxing is obtained by enabling a "security manager" inside the VM. The security manager will determine whether a particular operation should be permitted or rejected. Several high level mechanisms exist for role-based security and offer the possibility to sign trusted code for more elevated privileges.

When the VM bytecode is executed with the security manager set up, secured low level IO functions will have a hook, interrupting normal execution, and calling the corresponding check function from the security manager (for example calling the Java function DatagramSocket.send() will first call CheckConnect(String host, int port) if the security manager is enabled). This security function can possibly avoid calling the requested function by raising an exception.

The hooks are hardcoded directly inside the VM. This security mechanism is only able to allow/deny a call, not to rewrite the parameters (very useful to implement scratch file systems or to set up complex network environments).

3.9.1.2 Unix-like Security Mechanisms

The Unix operating system was designed to be portable, multi-tasking and multi-user in a time-sharing configuration. It has crystallized several concepts that shaped its design: the use of plain text for storing data; a hierarchical file system; treating devices and certain types of IPC (inter-process communication) as files. The latter is widely regarded as one of the defining points of Unix.

Considering most resources as files permits us to use file permissions, via users and groups, as the primary security mechanism in Unix-like operating systems. When a user executes an application, the corresponding process belongs to this user and inherits the same rights. Each successively executed process will in turn inherit the rights of the parent process. This is called DAC (discretionary access control): restricting access to objects based on the identity of subjects and/or groups to which they belong.

Over time, this initial security mechanism has been refined by many others which for the most part are specific to only some Unix variants.

chroot Literally 'change root', is a way to confine a process in a new virtual directory structure: the root directory of this structure is no more the root of the whole file system. The process has only access to the files available in this particular branch of the file system. Additionally to chroot, disk quotas permit to limit disk usage by user.

Mandatory Access Control (MAC) Several projects have added a MAC layer to the Linux kernel (and several other Unix variants). The most famous are probably SELinux and AppArmor. Both use LSM (Linux Security Module) interface to implements their security model. Their strategy is to finely define an exact set of resources that privileged or vulnerable programs need for a correct execution and deny everything else (least privilege policy).

seccomp seccomp is a simple sandboxing mechanism for the Linux kernel which allows a process to make a one-way transition into a "secure" state where the only system calls allowed are exit(), sigreturn(), read() and write() to already opened file descriptors. If the process attempt any other system calls, the kernel will terminate the process. It was originally intended as a means of safely running untrusted compute-bound programs and has recently been used by Google Chrome to sandbox the flash plugin and the rendering process of a web page.

3.9.1.3 Unix Isolation

Several Unix-like systems provide mechanisms to isolate an 'instance' of the current OS in a separate logical namespace. Concretely, the new system is initialized in a new namespace and all his child processes inherit it. Processes belonging to a namespace are then unable to see any processes not belonging to it. This isolation is done at the kernel level. Depending of the implementation, additional restrictions and priority for system resources can be given. Generally, the only way to access and use these instances is by using a dedicated IP, associated with a specific instance.

Some implementations of this scheme are Solaris Containers, FreeBSD Jails, Linux OpenVZ and Linux LXC.

This system permits us to keep excellent performance as the namespace restrictions are lightweight to apply. The main drawback is that the isolated guest system must be compatible with the host kernel.

3.9.1.4 Library Interposition

Library interposition is a mechanism to intercept a shared library function call. The interposition mechanism modifies the way the linker works by placing the new library first in the resolution list of the resolver.

This permits us to replace a function by another one, wrapping or denying access to the original one. However, library interposition is not an absolute security mechanism because a binary can still be statically linked (it directly includes its libraries) or do direct system calls (generally system calls are done using the `libc` that will do the mapping to the real system calls in order to ensure the compatibility of applications in the case where the kernel's system calls addresses or parameters change).

3.9.1.5 Emulation and Virtualization

Emulation provides a completely virtual hardware view. Any system able to run on that specific hardware can be installed. The emulated hardware does not need to share any characteristics with the host's hardware, but, when no optimizations based on hardware similarities are applicable, the overhead is very high.

Conversely, virtualization only provides virtual environments that run on the same hardware (host and guest systems need to be hardware compatible). This limitation provides a boost in performance because the virtualization mechanisms only 'protect' parts of the code (opcodes are filtered or altered to safely run in virtualization and to call back the virtualization manager when needed): most of it is directly executed by the physical CPU with no modifications.

The guest operating system image is portable as long the virtualization system is portable and the hardware compatible. The primary goal of virtualization is to maximize resource usage and not to act as a sandbox.

3.9.1.6 Language Level Sandboxing

Most scripting languages compile the source code to a specific VM code (this step is generally done transparently by an interpreter that includes a just-in-time compiler) and then execute it. The sandboxing environment can be added during the compilation process by using a modified compiler. Another solution is to implement the sandbox as part of the VM (see section 3.9.1.1). Finally the sandbox can be completely implemented by some languages: a specific running environment is set up before executing the application's code.

Only a minority of languages can provide "pure" language sandboxing. The main requirements are to have automatic memory management (memory allocation, garbage collector, no pointers, etc.) and the ability to modify every part of the environment (global functions, etc.). Additionally, the language must support a closure mechanism in order to save the original function and to be able to reuse it in the wrapping one.

Obviously, if the sandbox can be completely implemented in the language without having to maintain patches for the compiler or the interpreter, it will save a lot of time and improve its portability.

3.9.1.7 Summary

Table 3.1 summarizes the differences between these solutions.

3.9.2 The SPLAY Sandbox

After dismissing every OS specific protections, solutions that only works inside a guest OS or that requires administrator rights, the most pragmatic choice for SPLAY was to move towards a form of language sandboxing.

For SPLAY we also need to have a very fine granularity to sandbox network operations and also have the ability to rewrite calls instead of just blocking them. None of the actual VMs sandboxes provide these features (however as some are open source, we could have adapted them).

Before choosing Lua as our language for SPLAY, we also have considered using Ruby or Python.

Ruby is a multi purposes programming language. It supports multiple programming paradigms, including imperative, object-oriented, functional and reflective.

While being syntactically speaking one of the most expressive and powerful language, Ruby suffers from the poor performances of its interpreter, specially the threading part (it uses green threads, not native threads).

Ruby cannot be sandboxed easily because the existing low level functions cannot be replaced and there was no plan (or external projects) to provide a sandbox at the interpreter level.

	VMs	Barriers	Isolation	Library interposer	Virtualization and emulation	Language level
Transparency	yes	yes	yes	yes	yes	yes
Low overhead	yes[a]	yes	yes	yes	yes/no[b]	yes
Granularity	no	no	no	yes	no	yes
User level security	yes	yes/no[c]	no	no	no	yes
Portability	yes	no	no	no	yes	yes
Modularity	no	N/A[d]	yes	yes	no	yes
Alter function calls	no	no	no	yes	no	yes
Signed code support	yes	no	no	no[e]	no	no[e]
Different languages	yes	yes	yes	yes	yes	no[f]

[a] Hooks have a low *additional* overhead.

[b] In this case, the virtualization/emulation *is* the sandbox. Emulation has high overhead, virtualization a low one.

[c] It depends. Chroot can be done at user level, quotas cannot (and are user specific, not application).

[d] Each barrier is a specific restriction by itself.

[e] Can be implemented.

[f] The sandboxing language has to be used exclusively for the program (however it can be linked with third parties trusted libraries written in other languages).

Table 3.1 – Sandboxes comparison summary

Python supports multiple programming paradigms (primarily object-oriented, imperative, and functional). The language is popular and can mostly (but not completely as of version 2) be sandboxed at the language level. Some projects (such as [8]) exist to provide sandboxing directly at the compilation time.

Lua was the language that satisfied most of our needs and its propensity to be an embeddable language was very important for us in order to extend C libraries and keep a very low memory footprint.

3.9.3 Google NaCL (Native Client)

Google NaCl is an ongoing effort to provide an x86 native code sandbox [132] [1]. It was not available when the SPLAY project started, but for the sake of completeness, we will do a quick comparison between NaCl and SPLAY sandbox features.

Important characteristics of NaCl:

- Modified C compiler;
- Assembly code checking;
- System calls interception, arguments checking.

NaCl behaves like a small operating system with 44 system calls, that are supported similarly on Linux, MacOS and Windows. The same executable will run without modifications on all platforms.

NaCl being a specific environment, code has to be recompiled for it. NaCl code cannot access external hardware such as GPUs. Access could eventually be given through additional system calls, but this is unlikely.

Before running the code, the loader does a security scan on it. If some instructions/sequences of instructions are disallowed, the code is refused. The NaCl C compiler will produce code that will be accepted by the loader. Complex cases like self modifying

[1] In March 2010 an ARM implementation was released, then, as of March 2012, x86-64 and IA-32 are also supported. However all these implementations could only use code compiled to the host's native instruction set. A new project PNaCl (Portable Native Client) has been started to address this issue and to permit architecture-agnostic code using an intermediate bytecode representation from the LLVM compiler.

code will be rejected. The developers will be limited in providing obfuscated assembly code to NaCl.

NaCl communicates with the external world exclusively using system calls. That means that all the code has to be inside the sandbox, including all libraries that will be used. Multiple instances of the same NaCl application will for the same reason not be shared.

Advantages of NaCl:

- Language independent (everything that can be compiled/interpreted in C);
- Near native speed;
- The minimum memory footprint is smaller than SPLAY but, when including libraries, they are not shared between instances.

Advantages of the SPLAY sandbox:

- Shared memory between applications instances;
- Hardware architecture independent;
- OS independent;
- Easy to provide shared libraries inside the sandbox;
- Applications can communicate with the sandbox.

Overall, NaCl is very interesting given the fact that it permits application developers to reuse their favorite tools and languages by just replacing the final compiler.

NaCl could probably have been used by SPLAY if it had been available at the time the project started. However, adapting NaCL would have required significantly more work than the solution we have choosen.

3.10 SplayWeb

Operating of SPLAY via its traditional CLI (Command Line Interface) can still be too complicated for some users, especially for students that should maximize the time spent writing applications rather than learning how the testbed works.

Also, the CLI does not provide any form of user management. This also means that there are no limitations to execute jobs and no privacy of the resulting logs.

In order to address these limitations and to give access to our testbed both to our students and to people outside our university, we have developed a web interface called SplayWeb.

SplayWeb is a frontend for the Splay controller written using the Ruby on Rails framework. Its aim is to provide an authenticated access to a Splay swarm (multiple splayds spread around the world) in order to permit many users to use it as a shared overlay testbed.

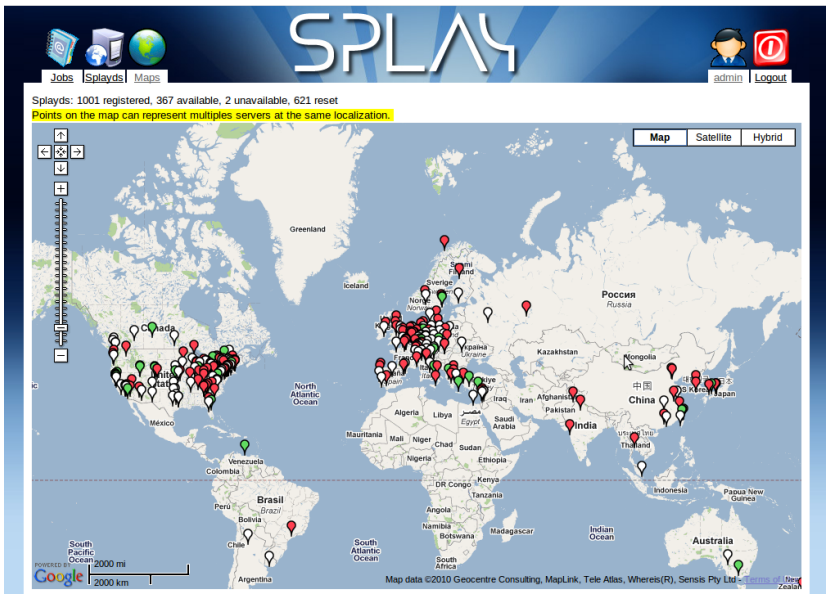


Figure 3.10 – Geolocalized world visualization of splayds running on PlanetLab.

It provides these new features to Splay:

User accounts An user can open an account to access the testbed provided by SplayWeb. In his account, he will be able to run its own experiments and get the results.

Map of nodes Geolocalization of nodes using their IPs. They are displayed on a Google Map (see Figure 3.10) for monitoring and also as a selection tool: one can visually choose a position on the map, a diameter and select only the splayds in that area for the next experiment.

Figure 3.11 – Configuring a new job.

Job submissing interface A user friendly interface (see Figure 3.11) is available to submit a new job (or to duplicate existing ones). The status of each job can be monitored. The logs of the jobs are directly available from the web interface while the experiment is running.

New splayd registration An interface is provided to register new splayds. They will be available for all users.

SplayWeb is already a tool of choice for students and guest users, but a future objective is to build a community of users. As PlanetLab do, in exchange of providing some splayds to the swarm, their owner will get time credits to run their own experiments.

3.11 Summary

Splay is an infrastructure that aims at simplifying the development, deployment and evaluation of large-scale distributed applications. It incorporates several novel features not found in existing tools and testbeds.

SPLAY applications are specified using in a high-level, efficient scripting language very close to the pseudo-code commonly used by researchers in their publications. They execute in a sandboxed environment and can thus be readily deployed on non-dedicated hosts. SPLAY also includes a comprehensive set of shared libraries tailored for the development of distributed protocols. Application specifications are based on an event-driven model and are extremely concise.

Unlike other solutions, it permits to reuse the same code to test both in isolated and real conditions. For the last, a particular attention has been given to the daemon, in order to run it efficiently on various dedicated and shared systems without compromising neither their performance or their security.

Chapter 4

Evaluation of SPLAY

> When in doubt, use brute force.
>
> Ken Thompson

4.1 Introduction

This chapter presents a thorough evaluation of SPLAY performance and capabilities. Evaluating such an infrastructure is a challenging task as the way users will use it plays an important role. Therefore, our goal in this evaluation is twofold: (1) to present the implementation, deployment and observation of real distributed systems by using SPLAY's capability to easily reproduce experiments that are commonly featured in research papers and (2) to study the performance of SPLAY itself, both by comparing it to other widely-used implementations and by evaluating its costs and scalability. The overall objective is to demonstrate the usefulness and benefits of SPLAY rather than evaluate the distributed applications themselves.

We first demonstrate in Section 4.2 SPLAY's capabilities to easily express complex system in a concise manner. We present in Section 4.3 the deployment and performance evaluation of the Chord DHT proposed in Section 3.7, using a ModelNet [127] cluster and PlanetLab [1]. We then compare in Section 4.4 the performance and scalability of the Pastry [110] DHT written with SPLAY against a legacy Java implementation, FreePastry [9]. Sections 4.5 and 4.6 evaluate SPLAY's ability to easily (1) deploy appli-

cations in complex network settings (mixed PlanetLab and ModelNet deployment) and (2) reproduce arbitrary churn conditions. Section 4.5 focuses on SPLAY performance for deploying and undeploying applications on a testbed. We conclude in Section 4.8 with an evaluation of SPLAY's performance with resource-intensive applications (tree-based content dissemination and long-term running of a cooperative Web cache). Featured experiences using SPLAY will also be detailed in chapters 5 and 6.

Experimental setup. Unless specified otherwise, our experimentations were performed either on PlanetLab, using a set of 400 to 450 hosts,[1] or on our local cluster. The cluster is composed of 11 nodes, each equipped with a 2.13 Ghz Core 2 Duo processor and 2 GB of memory, linked by a 1 Gbps switched network. All nodes run GNU/Linux 2.6.9. A separate node running FreeBSD 4.11 is used as a ModelNet router, when required by the experiment. Our ModelNet configuration emulates 1,100 hosts connected to a 500-node transit-stub topology. The bandwidth is set to 10Mbps for all links. RTT between nodes of the same domain is 10 ms, stub-stub and stub-transit RTT is 30 ms, and transit-transit (i.e., long range links) RTT is 100 ms. These settings result in delays that are approximately twice higher than those experienced in PlanetLab.

4.2 Development complexity

We present in this chapter the following applications using SPLAY:

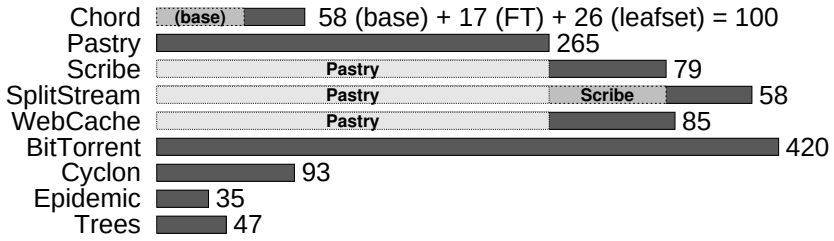
- Chord [121] and Pastry [110], two DHTs;
- Scribe [34], a publish-subscribe system;
- SplitStream [35], a bandwidth-intensive multicast protocol;
- A cooperative web-cache based on Pastry;
- BitTorrent [42], a content distribution infrastructure;[2]
- Cyclon [129], a gossip-based membership management protocol.

[1] SPLAY daemons have been continuously running on these hosts for more than one year.

[2] Note that, without the requirement for binary compatibility, the size of our implementation could be significantly reduced. Our BitTorrent implementation has been successfully used for downloading several times the Ubuntu Linux distribution in official swarms.

We have also implemented a number of classical algorithms, such as epidemic diffusion on Erdös-Renyi random graphs [47] and various types of distribution trees [25] (n-ary trees, parallel trees).

As one can note from the following figure, all implementations are extremely concise in terms of lines of code (LOC). Number and darker bars represent LOC for the protocol, while lighter bars represent protocols acting as a substrate (i.e., Scribe is based on Pastry, and SplitStream is based on both Pastry and Scribe):[1]



Although the number of lines is clearly just a rough indicator of the expressiveness of a system, it is still a valuable metric to estimate programming efforts. Our implementations are systematically more compact than those written with Mace [72] (by approximately a factor of two) and comparable to P2's [89] specifications. A well-documented protocol such as Chord only took a few hours to implement and debug. In contrast, BitTorrent, being a complex and underspecified protocol, required several days of development. In both cases, the development process greatly benefited from the short deployment and testing phase, made almost trivial by SPLAY.

4.3 Testing the Chord Implementation

This section presents the deployment and performance results of the Chord implementation from Section 3.7. We proceed with two deployments. First, the exact code presented in this chapter is deployed in a ModelNet testbed with no node failure. Second, a slightly modified version of this code is run on PlanetLab. This version includes

[1] Note that we did not try to compact the code in a way that would impair readability. All lines have been counted, including those that only contain block delimiters.

the extensions presented at the end of Section 3.7: use of a leaf set instead of a single successor and a single predecessor, fault-tolerant RPCs, and shorter stabilization intervals.

4.3.1 Chord on ModelNet

To parameterize the deployment of the Chord implementation presented in Section 3.7 on a testbed, we create a descriptor that describes resources requirements and limitations. The descriptor allows to further restrict memory, disk and network usage, and it specifies what information an application should receive when instantiated:

```
--[[ BEGIN SPLAY RESOURCES RESERVATION
  nb_splayd  1000
  nodes      head    1
END SPLAY RESOURCES RESERVATION ]]
```

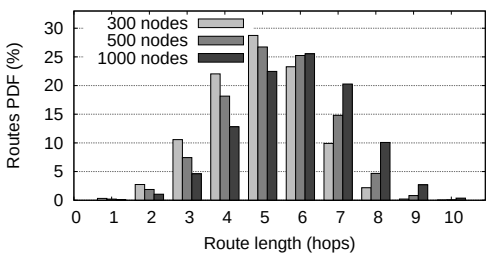
This descriptor requests 1,000 instances of the application and specifies that each instance will receive three essential pieces of information: (1) a single-element list containing the first node in the deployment sequence (to act as rendezvous node); [1] (2) the rank of the current process in the deployment sequence; and (3) the identity of the current process (host and port). This information is useful to bootstrap the system without having to rely on external mechanisms such as a directory service. In the case of Chord, we use this information to have hosts join the network one after the other, with a delay between consecutive joins to ensure that a single ring is created. The following code is added to the Chord code:

```
events.sleep(job.position)                    -- 1s between joins
if #job.position > 1 then                     -- first node is rendez-vous node
  join(job.nodes(1))
end
```

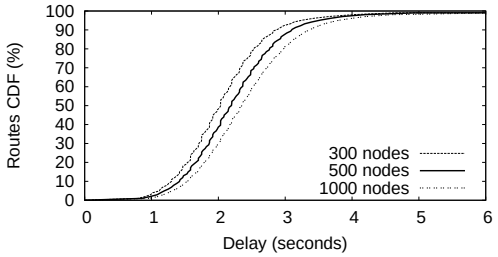
Finally, we register the Lua script and the deployment descriptor using one of the command line, web services or the web-based interfaces.

Each host runs 27 to 91 Chord nodes (we show in Section 4.4 that SPLAY can handle many more instances on a single host). During the experiment, each node injects 50 random lookup requests in the system. We then undeploy the overlay, and process the results obtained from the logging facility. Figure 4.1a presents the distribution of route

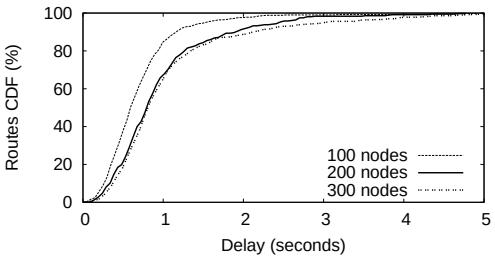
[1] Bootstrap information contain a selected number of nodes from the complete list. The selection of this subset can be the head of the list or random elements.



(a) ModelNet: route lengths



(b) ModelNet: delays



(c) PlanetLab: delays

Figure 4.1 – Performance results of Chord, deployed on a ModelNet cluster and on PlanetLab.

lengths. Figure 4.1b presents the cumulative distribution of latencies. The average number of hops is below $\frac{\log_2 N}{2}$ and the look-up time remains small. This prefigures our observations that SPLAY is efficient and does not introduce additional delays or overheads.

4.3.2 Chord on PlanetLab

Next, we deploy our Chord implementation with extensions on 380 PlanetLab nodes and compare its performance with MIT's finely-tuned C++ Chord implementation [10] in terms of delays when looking up random keys in the DHT. In both cases, we let the Chord overlay stabilizes before starting the measurements.

Figure 4.2 presents the cumulative distribution of delays for 5000 random lookups (average route length is 4.1 for both systems). We observe that MIT Chord outperforms Chord for SPLAY, because it relies on a dedicated network layer that uses, amongst other optimizations, network coordinates for constructing latency-aware finger tables. In contrast, we did not include such optimizations in our implementation. If SPLAY allows to quickly prototype and evaluate algorithms, it does no magic in tuning and enhancing protocols: it just helps designers in the process.

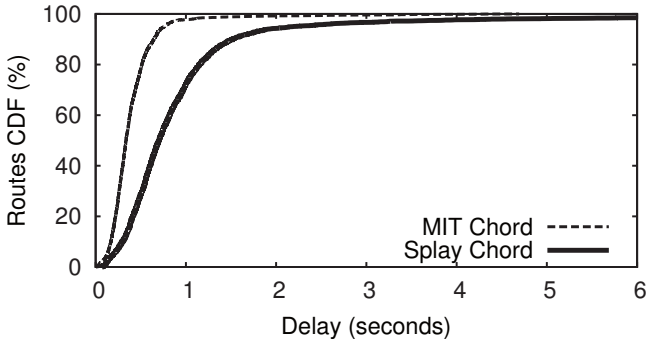


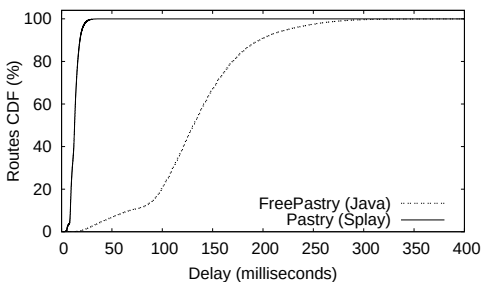
Figure 4.2 – Comparison of delays of MIT Chord and SPLAY Chord on PlanetLab

4.4 SPLAY Performance

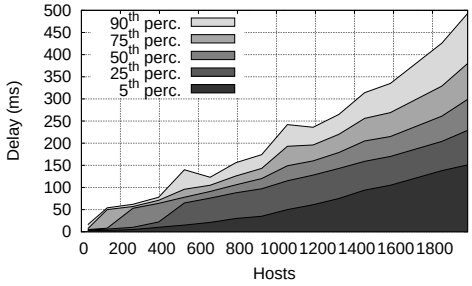
We evaluate the performance of applications using SPLAY in two ways. First, we evaluate the efficiency of the network libraries, based on the delays experienced by a sample application on a high-performance testbed. Second, we evaluate scalability: how many nodes can be run on a single host and what is the impact on performance. For these tests we chose Pastry [110] because: (i) it combines both TCP and UDP communications; (ii) it requires efficient network libraries and transport layers, each node being potentially opening sockets and sending data to a large number of other peers; (iii) it supports network proximity-based peer selection, and as such can be affected by fluctuating or instable delays (for instance due to overload or scheduling issues). Our Pastry implementation has been intentionally developed without optimizations not documented in [110] to allow for a fair evaluation.

We compare our version of Pastry with FreePastry 2.0 [9], a complete implementation of the Pastry protocol in Java. Our implementation is functionally identical to FreePastry and uses the very same protocols, e.g., locality-aware routing table construction and stabilization mechanisms to repair broken routing table entries. The only notable differences reside in the message formats (no wire compatibility) and the choice of alternate routes upon failure.

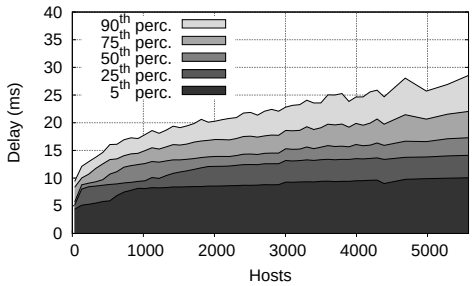
We deployed FreePastry using all optimizations advised by the authors, that is, running multiple nodes within the same JVM, replacing Java serialization with raw serialization, and keeping a pool of opened TCP connections to peers to avoid reopening recently used connections. We used 3 JVMs on our dual cores machines, each running multiple Pastry nodes. With large set of nodes, our experiments have shown that this configuration yields slightly better results than using a single JVM, both in terms of delay and load.



(a) Delay distribution comparison, 980 nodes



(b) FreePastry, evolution of delays



(c) Pastry for SPLAY, evolution of delays

Figure 4.3 – Comparisons of two implementations of Pastry: FreePastry and Pastry for SPLAY.

Figure 4.3a presents the cumulative delay distribution in a converged Pastry ring. The distribution of route lengths (not shown) is slightly better with FreePastry thanks to optimizations in the routing table management. Delays obtained with Pastry on SPLAY are much lower than the delays obtained with FreePastry. This experiment shows that SPLAY, while allowing for concise and readable protocol implementations, does not trade simplicity for efficiency. We also notice that Java-based programs are often too heavyweight to be used with multiple instances on a single host. This is further conveyed by our second experiment that compares the evolution of delays of FreePastry (Figure 4.3b) and Pastry for SPLAY (Figure 4.3c) as the number of nodes on the testbed increases. We use a percentile-based plotting method that allows expressing the evolution of a cumulative distribution of delays with respect to the number of nodes. We can observe that: (1) delays start increasing exponentially for FreePastry when there are more than 1,600 nodes running in the cluster, that is 145 nodes per host (recall that all nodes on a single host are hosted by only 3 JVMs and share most of their memory footprint); (2) it is not possible to run more than 1,980 FreePastry nodes, as the system will start swapping, degrading performance dramatically; (3) SPLAY can handle 5,500 nodes (500 on each host) without a significant drop in performance (other

than the $\mathcal{O}(\log N)$ route sizes evolution, N being the number of nodes).

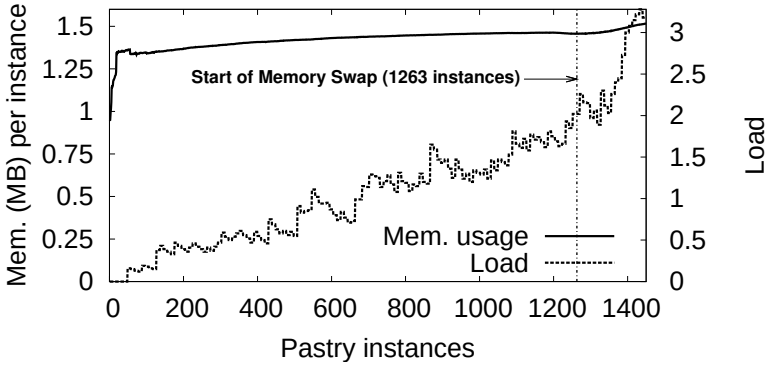


Figure 4.4 – Memory consumption and load evolution on a single node hosting several instances of Pastry for SPLAY.

Figure 4.4 presents the load (i.e., the average number of processes with "runnable" status, as reported by the Linux scheduler) and memory consumption per instance for varying number of instances. Each process is a Pastry node and issues a random request every minute. We observe that the memory footprint of an instance is lower than 1.5 MB, with just a slight increase during the experiment as nodes fill their routing table. It takes 1,263 Pastry instances before the host system starts swapping memory to disk. Load (averaged over the last minute) remains reasonably low, which explains the small delays presented by Figure 4.3c.

4.5 Complex Deployments

SPLAY is designed to be used within a large set of different testbeds. Despite this diversity, it is sometimes also desirable to experiment with more than a single testbed at a time. For instance, one may want to evaluate a complex system with a set of peers linked by high bandwidth, non-lossy links, emulated by ModelNet, and a set of peers facing adverse network conditions on PlanetLab. A typical usage would be to test a broker-based publish-subscribe infrastructure deployed on reliable nodes, along with a set of client nodes facing churn and lossy network links.

Such a mixed deployment is hard to set up using scripting and common tools, as one has to care about NAT and firewalls traversal, port forwarding, etc. The experiment presented in this section shows that such a complex mixed deployment can be achieved

using SPLAY as if it were on a single testbed. The only precondition is that the administrator of a testbed behind a NAT or firewall defines (and opens) a range of ports that all `splayd`s will use to communicate with other daemons outside the testbed. All other communication details are dealt with by SPLAY itself: no modification is needed to the application code.

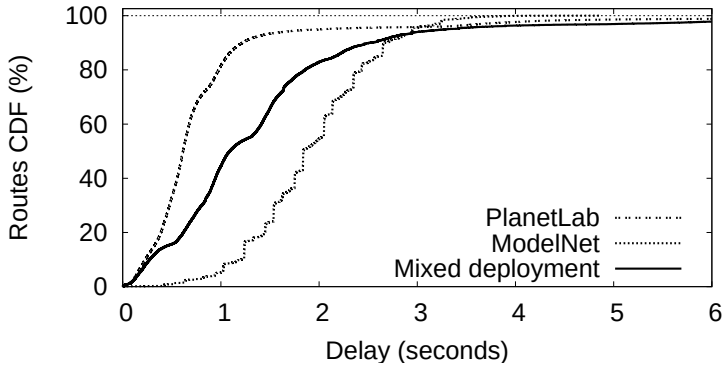


Figure 4.5 – Pastry on PlanetLab, ModelNet, and both.

Figure 4.5 presents the delay distribution for a deployment of 1,000 nodes on PlanetLab, on ModelNet, and in a mixed deployment over both testbeds at the same time (i.e., 500 nodes on each). We notice that the delays of the mixed deployment are distributed between the delays of PlanetLab and the higher delays of our ModelNet cluster. The "steps" on the ModelNet cumulative delays representation are a result of routes of increasing number of hops (both in Pastry and in the emulated topology), and the fixed delays for ModelNet links.

4.6 Using Churn Management

This section evaluates the use of the churn management module, both using traces and synthetic descriptions. Using churn is as simple as launching a regular SPLAY application with a trace file as an extra argument. SPLAY provides a set of tools to generate and process trace files. One can, for instance, speed-up a trace, increase the churn amplitude whilst keeping its statistical properties, or generate a trace from a synthetic description.

Figure 4.6 presents a typical experiment of a massive failure using the synthetic description. We ran Pastry on our local cluster with 1,500 nodes and, after 5 minutes,

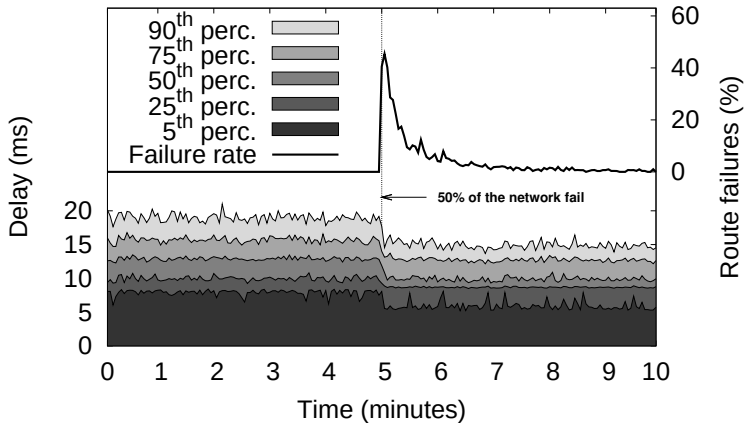


Figure 4.6 – Using churn management to reproduce massive churn conditions for the SPLAY Pastry implementation.

triggered a sudden failure of half of the network (750 nodes). This models, for example, the disconnection of a inter-continental link or a WAN link between two corporate LANs. We observe that the number of failed lookups reaches almost 50% after the massive failure due to routing table entries referring to unreachable nodes. Pastry recovers all its routing capabilities in about 5 minutes, and we can observe that delays actually decrease after the failure because the population has shrunk (delays are shown for successful routes only). While this scenario is amongst the simplest ones, churn descriptions allow users to experiment with much more complex scenarios, as discussed in Section 3.8.

Our second experiment is representative of a complex test scenario that would usually involve much engineering, testing and post-processing. We use the churn trace observed in the Overnet file sharing peer-to-peer system [23]. We want to observe the behavior of Pastry, deployed on PlanetLab, when facing churn rates that are much beyond the natural churn rates suffered in PlanetLab. As we want to increase levels of Churn, we simply "speed-up" the trace, that is, with a speed-up factor of 2, 5 and 10 a minute in the original trace represents 30, 12 and 6 seconds respectively.

Figure 4.7 presents both the churn description and the evolution of delays and failure rates, for increasing levels of churn. The churn description shows the population of nodes and the number of joins/leaves as a function of time, and performance observations plot the evolution of the delay distribution as a function of time. We observe that (1) Pastry handles churn pretty well as we do not observe a significant failure rate when as much as 14% of the nodes are changing state within a single minute; (2) running this experiment is neither more complex nor longer than on a single cluster

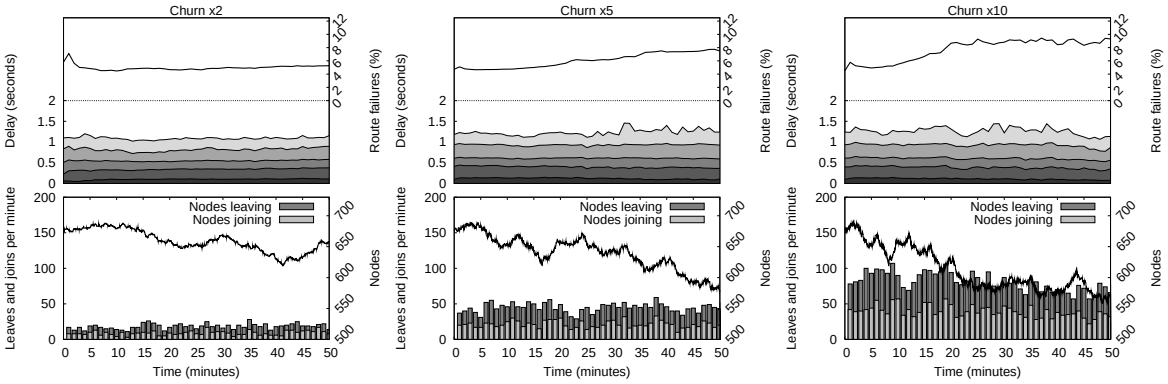


Figure 4.7 – Study of the effect of churn on Pastry deployed on PlanetLab. Churn is derived from the trace of the Overnet file sharing system and sped up for increasing volatility.

without churn, as we did for Figure 4.3a. Based on our own experience, we estimate that it takes at least one order of magnitude less human efforts to conduct this experiment using SPLAY than with any other deployment tools. We strongly believe that the availability of tools such as SPLAY will encourage the community to further test and deploy their protocols under adverse conditions, and to compare systems using published churn models.

4.7 Deployment Performance

This section presents an evaluation of the deployment time of an application on an adversarial testbed, PlanetLab. This further conveys our position from Section 3.2 that one needs to initially select a larger set of nodes than requested to ensure that one can rely on reasonably responsive nodes for deploying the application. Traditionally, such a selection process is done by hand, or using simple heuristics based on the load or response time of the nodes. SPLAY relieves the need for the user to proceed with this selection.

Figure 4.8 presents the deployment time for the Pastry application on PlanetLab. We vary the number of additionally probed daemons from 10% to 100% of the requested nodes. We observe that a larger set results in lower delays for deploying an application (hence, presumably, lower delays for subsequent application communications). Nonetheless, the selection of a reasonably large superset for a proper selection of peers is a tradeoff between deployment delay and redundant messages sent over the network.

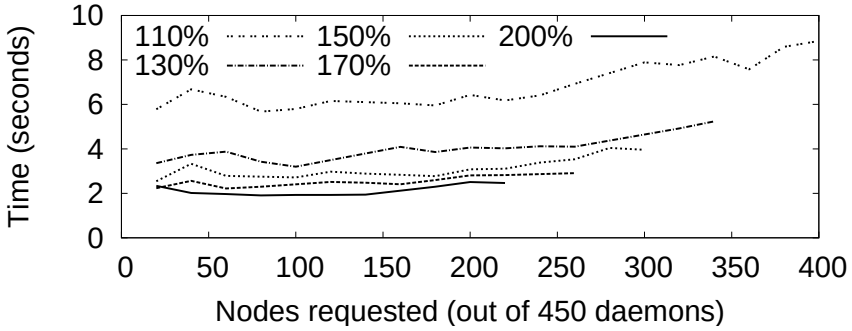


Figure 4.8 – Deployment times of Pastry for SPLAY, as a function of (1) the number of nodes requested and (2) the size of the superset of daemons used.

4.8 Bandwidth-intensive Experiments

Our two last experimental demonstrations deal with resource-intensive applications, both for short-term and long-term runs. They further conveys SPLAY's ability to run in high performance settings and production environments, as well as demonstrating that the obtained performance is similar to the one achieved with a dedicated implementation (particularly from the network point of view). We run the following two experiments: (1) the evaluation of a cooperative data distribution algorithm based on parallel trees using both SPLAY and a native C implementation on ModelNet and (2) a distributed cooperative Web cache for HTTP accesses, which has been running for several weeks under a constant and important load.

4.8.1 BitTorrent dissemination.

BitTorrent is a peer-to-peer application for distributing large files. We have developed a BitTorrent implementation for SPLAY that is binary compatible with other clients, i.e., our implementation can join an existing BitTorrent swarm.[1] We evaluate our application on PlanetLab by distributing a 16 MB file split in 128 blocks of 128 kB. A single source is used and peers that have finished downloading remain in the network. Figure 4.9 presents the cumulative delay for the reception of the entire file, for populations of 100 and 300 peers. Download rates range from 30.72 KB/s to 1.536 MB/s. The evolution of the file transfer behaves as expected, with the number of completed

[1]Note that, without the requirement for binary compatibility, the size of our implementation could be significantly reduced.

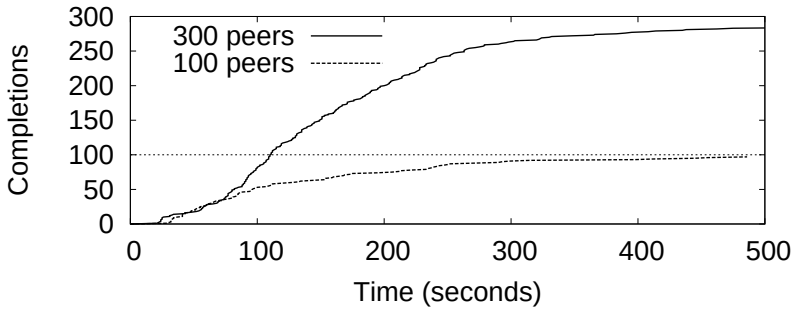


Figure 4.9 – Distribution of a 16 MB file using the BitTorrent SPLAY implementation.

peers growing steadily in the beginning and flattening when only slow peers remain. Although bandwidth experiments on PlanetLab are known to be non-reproducible due to the ever-changing load conditions, one can note that out results are consistent with other studies of BitTorrent (e.g., [24]).

Nodes with the highest bandwidth complete after 10 to 20 seconds. Thereafter, the rate is 1 peer completing the download per second, while problematic peers (overloaded peers, on which bandwidth is shared among several PlanetLab experiments)

We observe that the fastest nodes (most likely those with the best bandwidth) complete after approximately 10 to 20 seconds. Thereafter, as the blocks are well spread throughout the network, the completion rate is quite high (one completion per second). After the first 50 peers, the rate slows down for the experiment with 100 clients while its keeps steady for 300 clients; only after 200 peers does it slow down. This can be explained by the fact that a number of nodes have limited bandwidth (or had an erratic behavior during the experiments) and take much longer to complete.[1] We observe the same faulty nodes in both experiments, a default that is due to the testbed (PlanetLab) and not to SPLAY itself.

4.8.2 Dissemination using trees

This experiment compares two versions of a simple cooperative protocol [25] based on parallel n-ary trees written with SPLAY, and in C. We create $n = 2$ distinct trees in the same manner as SplitStream [35] does: each of the 63 nodes is an inner member in one tree and a leaf in the other. The data to be transmitted is split into blocks, which are propagated along one of the 2 trees according to a round-robin policy. This experiment

[1] In several experiments, we observed that a few nodes failed and never completed their download.

allows us to observe how SPLAY compares against a native application, CRCP, written in C [11]. Using a tree for this comparison bears the advantage of highlighting the additional delays and overheads of the platform and its network libraries (such as the sandboxing of network operations). These overheads cumulate at each level of the tree, from the root to the leaves.

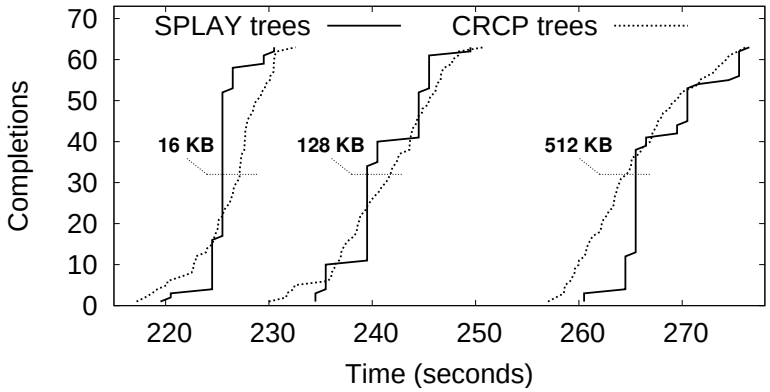


Figure 4.10 – File distribution using trees.

Tests were run in a ModelNet testbed configured with a symmetric bandwidth of 1 Mbps for each node. Results are shown in Figure 4.10 for binary trees, a 24 MB file, and different block sizes (16 KB, 128 KB, 512 KB). We observe that both implementation produce similar results, which tends to demonstrate that the overhead of SPLAY's language and libraries is negligible. Differences in shape between CRCP and SPLAY are due to CRCP nodes sending chunks sequentially to their children, while SPLAY nodes send chunks in parallel. In our settings (i.e., homogeneous bandwidth), this should not change the completion time of the last peer as links are saturated at all times.

4.8.3 Long-running experiment: cooperative Web cache

Our last experiment presents the performance over time of a cooperative Web cache built using SPLAY following the same base design as Squirrel [61]. This experiment highlights the ability of SPLAY to support long-run applications under constant load. The cache uses our Pastry DHT implementation deployed in a cluster, with 100 nodes that proxy requests and store remote Web resources for speeding up subsequent accesses. For this experiment, we limit the number of entries stored by each nodes to 100. Cached resources are evicted according to an LRU policy or when they are older than 120 seconds.

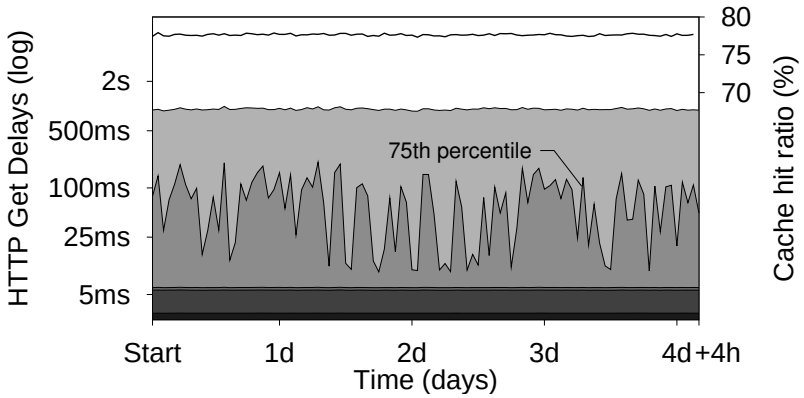


Figure 4.11 – Cooperative Web cache: evolution of delays and cache hit ratios during a 4 days period.

The cooperative Web cache has been run for three weeks. Figure 4.11 presents the evolution of HTTP requests delay distribution for a period of 100 hours along with the cache hit ratio. We injected a continuous stream of 100 requests per second extracted from real Web access traces [12] corresponding to 1.7 million hits to 42,000 different URLs. We observe a steady cache hit ratio of 77.6%. The experienced delays distribution has remained stable throughout the whole run of the application. Most accesses (75th percentile) are cached and served in less than 25 to 100 ms, compared to non-cached access that require 1 to 2 seconds on average.

4.9 Summary

SPLAY can seamlessly deploy applications in real (e.g., PlanetLab) or emulated (e.g., ModelNet) networks, as well as mixed environments. An original feature of SPLAY is its ability at injecting churn in the system using a trace or a synthetic description to test application in the most realistic conditions.

Our thorough evaluation of SPLAY demonstrates that it allows developers to easily express complex systems in a concise yet readable manner, scales remarkably well thanks to its low footprint, exhibits very good performance in various deployment scenarios, and compares favorably against native applications in our experiments.

Chapter 5

PULP

> The biggest difference between time and space is that you can't reuse time.
>
> M. Furst

5.1 Introduction

Disseminating information from and to very large communities of nodes is fundamental in many systems and for a wide spectrum of applications, such as spreading antivirus updates, propagating control messages or monitoring information (particularly when arriving in bursts), operating a content delivery network, etc.

Solutions based on *dedicated resources*, be they individual servers, server farms, or distributed architectures of dedicated forwarders, share the common issue of *cost effectiveness*, as they are extremely difficult to provision accurately. More specifically, the almost inevitable *over-provisioning of resources* raises cost dramatically. Any amount of dedicated resources has a finite limit on the load it can handle. A system claiming high scalability should provision resources for the highest possible anticipated load, even if that peak load appears very rarely, if at all. This results in a significantly underutilized system during regular operation, leading to massive waste of resources and money.

Collaborative architectures embracing peer-to-peer (P2P) models form the natural alternative to systems based on dedicated resources. Nodes making use of a service also contribute to it, relieving the total system of part of the work in a proposition comparable to the load they impose on it. This property is often referred to as *elasticity*, in the sense that system resources rapidly scale up and down along with the demand.

In this context, *epidemic* (or *gossip*) *protocols* have recently received an increasing interest. Their attractiveness stems from their scalability, inherent balancing of load across all nodes, and quick convergence. Additionally, they have proven to be remarkably robust in the face of failures. Last but not least, they are extremely simple. They rely on a periodic pairwise exchange of information between peers, and are particularly suited to implement a global emergent behavior as a result of local interactions based on limited knowledge. They have been applied to a wide variety of applications [45, 63, 67, 69, 106]. Yet, the most classical use of gossip protocols is to reliably disseminate data in large networks in a collaborative manner [26, 48, 68].

Gossip-based dissemination transmits information in the same way as a rumor spreads within a large group of people in real life, or a disease spreads by infecting members of a population, which can in turn infect others. Randomness and repetitive probabilistic exchanges are at the heart of gossip-based protocols and are keys to achieve robustness. Messages are relayed in an epidemic manner so that, with high probability, they are received by all peers in the system.

More recently, gossip-based approaches have been extensively used in the context of streaming applications [80, 81, 133]. Some recent work [28] has demonstrated that epidemic live streaming algorithms can achieve nearly unbeatable rates and delays. This growing interest in gossip-based dissemination can be explained by the more than ever dynamic nature of large-scale systems with frequent failures and unreliable communication links, emphasizing the fragile nature of tree-based approaches. These systems consider the transmission of a stream of messages from one source to a large number of consumers, typically for in-order replay. However, besides the now well-understood single-source streaming problem and associated protocols, there is also a need from applications for the support of all-to-all data transmission where messages are emitted by potentially all nodes in the system and shall be received by all others.

In this context, message emissions are generally not correlated, thus a weak or no ordering is typically sufficient. Examples of such applications include wide-area monitoring, logging and update mechanisms and notification services. These applications are the ones we are targeting in this chapter.

They share the following common characteristics. Messages are sent by multiple sources (i.e., any peer in the system can be the source of a new message or set of messages).

Messages are typically of moderate size and do not need to be sent in several pieces (or chunks), which would typically be achieved in a single-source dissemination using protocols such as BitTorrent [42].

While messages from different sources are not necessarily correlated, the overall rate at which messages are sent in the whole group is typically varying in time: burst of messages can be sent to all peers in the system at some point in time while in the common case only a few messages are disseminated per unit of time, or messages can be triggered at multiple sources in response to the reception of a previous message. Finally, the targeted infrastructure is a non-reliable one, where messages can be lost and nodes can join or leave the system at any moment. A dissemination service must take this aspect into account but at the same time achieve reliable and efficient dissemination at the lowest possible cost on the network (by minimizing the number of messages and the overall bandwidth used for the dissemination and its management).

There are two main methods for epidemic dissemination of messages, as laid out in the seminal paper on epidemics by Demers et al. [45]. The first one (referred to as *rumor mongering* in [45]) is a *reactive* method. Upon reception of a new message, a node actively pushes it forward to a few other nodes in the network, which, in turn, do the same until some termination condition is met. The second method (referred to as *anti-entropy* in [45]) is *proactive*. Each node periodically probes a random other node to check for messages it has not received yet, and pulls them if there are any. For convenience, in this chapter we will be referring to these methods as *push* and *pull*, respectively.

PULP has been completely realized and tested using SPLAY. The complete implementation consists of 400 lines of Lua code, including the underlying Cyclon [129] implementation. It has been a good showcase for SPLAY in order to demonstrate its efficiency to develop new protocols from scratch and its ability to easily reuse already developed components such as Cyclon.

5.1.1 Evaluation Metrics and Objectives

In order to evaluate the strengths and weaknesses of gossip-based dissemination protocols, we consider a set of metrics traditionally used to assess the performance of dissemination protocols, namely *delay*, *coverage*, and *redundancy*.

- **Delay** refers to the time intervening between the generation of a message and its delivery at some destination.

- **Coverage** refers to the ratio of peers that receive a message. We are targeting coverage ratios of 100%, i.e., each message should be delivered to *all* peers.

- **Redundancy** refers to the number of duplicate—and therefore unnecessary—message deliveries. Although it improves resilience to failures, too high a redundancy may overload the network.

Another important metric is that of the overall cost of the dissemination mechanism, in terms of the number of messages (for the dissemination itself, maintenance of the dissemination substrate, and control flow messages) and used bandwidth (typically dominated by redundant sends of messages). Ideally, messages originating at any peer should be delivered to *all* peers in the system (complete coverage) with reasonable delays. In our context, this should be achieved in a robust manner and with the lowest possible cost, meaning achieving a minimal redundancy and using as few messages as possible for the protocol operation.

5.1.2 Contributions

We start by pointing out the shortcomings of the push and the pull methods, and illustrate them by experimental results. Our contributions are then the following. We present the specificities related to the dissemination of a *flow of messages* and how forwarding at random directions can be leveraged to achieve *complete* disseminations at low cost and with low delay. Next, we present a new protocol based on our observations, PULP, that mingles push and pull in such a way that each one makes up for the weaknesses of the other. PULP is a highly scalable and adaptable collaborative protocol for the dissemination of multiple messages in very large sets of peers. The performance, costs, and resilience of PULP are conveyed by real deployments under static and dynamic scenarios on a cluster and on the PlanetLab testbed [1] using the SPLAY framework.

5.1.3 Outline

The rest of the chapter is organized as follows: In Section 5.2, we discuss the strengths and weaknesses of the push and pull approaches. The design of the PULP protocol is presented in Section 5.3. In Section 5.4, we perform a thorough experimental evaluation of PULP. Section 5.5 discusses related work. Finally, Section 5.6 concludes.

5.2 The Push-Pull Dilemma

As already mentioned, epidemic-based dissemination protocols are based on two basic methods: *push* and *pull*. In this section, we identify the strengths and weaknesses of each approach in an attempt to come up with an efficient hybrid scheme that combines the best of both worlds.

5.2.1 Push protocols

Push protocols are based on the recursive forwarding of messages among peers. A node receiving a message actively passes it on to a few random other nodes, which recursively do the same until some termination condition is met. The termination condition ensures that the recursion does not go on forever. For instance, messages could be augmented by a Time-to-Live (TTL) field to limit the number of hops they can take. Alternatively, nodes could be programmed to forward messages only upon their first reception and ignore subsequent copies. Either solution ensures that the dissemination of a message eventually fades out. The number of times an informed node forwards a message is denoted as the FANOUT.

Regardless of the specific variation of the push protocol, reaching *all* nodes by blindly forwarding a message in random directions is a very expensive operation. Assume a rather ideal and generic model for push dissemination (we explain later why reality is harsher), where nodes are selected *uniformly at random* and *one at a time*, out of a total population of n nodes. At each iteration, the message is forwarded to the selected node, independently of whether it is already informed.

The probability to select a not-yet-informed node when k nodes have already been informed is $\frac{n-k}{n}$, which requires an expected number of $\frac{n}{n-k}$ random forwards to reach the $(k+1)$-th node. This number lies between one and two until half of the nodes have been informed, but increases dramatically for the last few nodes. For instance, reaching the *last* node alone requires on average n forwards. The expected number of times a message should be forwarded to reach the whole population of n nodes is

$$\sum_{k=0}^{n-1} \frac{n}{n-k} = n \cdot \sum_{i=1}^{n} \frac{1}{i} \approx n \ln n + \gamma n$$

for high values of n, where $\gamma \approx 0.5772$. That is, the expected total number of forwards to reach all nodes is in the order of $\mathcal{O}(n \ln n)$.[1]

[1]These probabilities are studied in the equivalent *Coupons Collector* problem, where a collector

Reality, however, is harsher. First, forwarding a message $n \ln n$ times *does not ensure* that it will reach all nodes. Second, as dissemination is carried out in a totally decentralized, asynchronous, and massively parallel way, it is not possible to impose fine-grained control on the number of times a message is forwarded. As a result of these, the indicated value $n \ln n$ often ends up being significantly surpassed, resulting in significant additional redundancy.

5.2.2 Pull protocols

In a pull protocol, each node periodically probes random peers in the network in hope to reach an already informed peer, and retrieves new messages when available. Typically, during a pull round, random pairs of peers exchange information about the messages they have recently received and request missing messages from each other.

Contrary to push protocols, the probability of an uninformed node receiving a message increases linearly with the current coverage of the message. Indeed, if k out of n nodes are informed, then a non-informed node will probe an informed one with probability $\frac{k}{n-1}$. In the case of the *last* uninformed node, it will pull the message with probability 1 the next time it probes a random node, as all other nodes already have it.

5.2.3 Coverage versus redundancy

In order to experimentally validate the aforementioned properties, we consider the following push protocol. Each message is augmented by a TTL value, determining the number of hops it can traverse. A node receiving a message for the *first time* decreases its TTL by one, and if it is not lower than zero forwards the message to FANOUT random other nodes. When a node receives a message that it has already received (and possibly forwarded) in the past, it simply ignores it.

Figure 5.1 presents a simulated overview of the behavior of push- and pull-only protocols, with respect to coverage and redundancy. For the sake of simplicity, both protocols are presented in a synchronous way: all peers that pull from, or push to, another node do so at the same time. This synchronous activity is called a *cycle*. We consider FANOUT values of 2 (top) and 4 (middle), with infinite TTL in push protocols. A random node sends a message at cycle 0.

keeps selecting at random out of n different coupons with replacement, and the number of trials until all coupons have been selected at least once is measured.

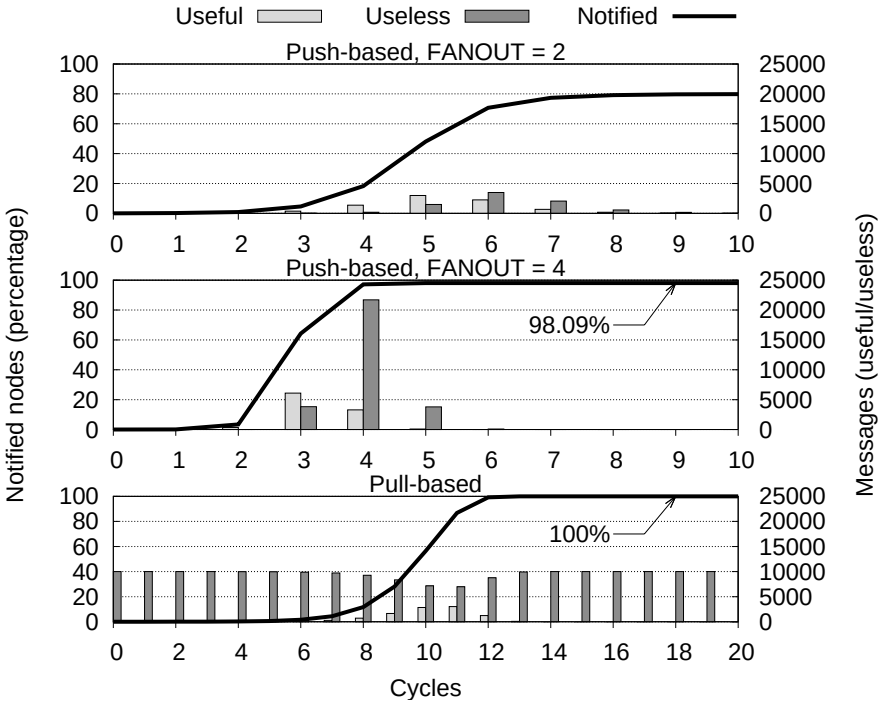


Figure 5.1 – Discrete time simulation of redundant ("useless") message delivery ratios and coverage for push-based and pull-based epidemic diffusions in a 10,000 node network. TTL is ∞ for the push-based simulation.

We observe that in push protocols the data spreads exponentially fast through the network, especially in the first rounds. However, as rounds advance, the rate of the dissemination diminishes and the cost of reaching additional nodes increases drastically (as shown by the number of redundant messages, i.e., messages pushed to already informed nodes). A higher FANOUT produces a sharper exponential growth of the set of reached nodes, but also a higher level of redundancy. One can notice that messages are not pushed to all nodes.

Regarding pull protocols, Figure 5.1 (bottom) shows that it takes several rounds until the dissemination of a message starts taking off, but once it has reached a sufficient number of nodes it quickly spreads to *all* the remaining ones. One can also observe that a constant number of control messages are sent during each round, with the ratio of useful messages growing only during the peak of the message spread.

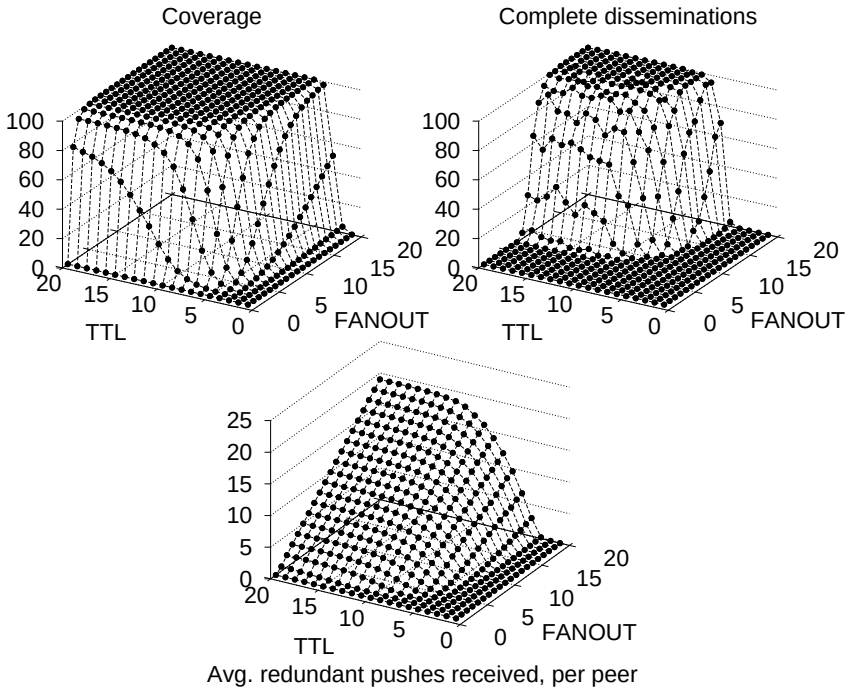


Figure 5.2 – Push-only: Coverage, number of complete disseminations and average number of redundant (useless) messages received per peer, as a function of the TTL and FANOUT.

Figure 5.2 sheds more light on the behavior of push protocols, with each dot representing a separate run of a push-only dissemination for a certain combination of TTL and FANOUT values. Even if a coverage of close to (but less than) 100% can be achieved

with a relatively low TTL and FANOUT, as illustrated in Figure 5.2 (top left), the probability of a message reaching *all* nodes is practically zero, unless both parameters have substantially higher values (top right). Unfortunately, the values for TTL and FANOUT that provide good coverage properties produce high redundancy, expressed as the number of deliveries per message per node (bottom).

5.2.4 Delay

Push protocols deliver messages with a relatively low delay, as they can immediately push messages further upon reception at each step of the dissemination, in an avalanche-like manner. Latency is therefore directly proportional to the network delays and the number of hops from the source. In contrast, pull protocols can exhibit high latency because, even if the propagation time of messages does also depend on the number of hops from the source, it is multiplied by the pull period. A high frequency will produce much (unnecessary) traffic while a low frequency will dramatically increase latency.

5.2.5 Discussion

While both push and pull protocols may achieve full coverage, they do so through complementary patterns. Push quickly spreads messages to a large portion of the network, as it does not depend on timing assumptions (e.g., no periodic operation). It is, however, slow and prohibitively expensive in reaching the last few nodes. This renders it an excellent candidate for the early stages of dissemination, but inappropriate for the final phase.

Pull, on the contrary, is an excellent candidate for the final stages of dissemination, as it deterministically delivers each message to *all* remaining nodes in logarithmic steps, and by pulling messages selectively it eliminates the problem of redundant message forwarding. However, it is a poor choice for the early stages, as it starts very slowly. The main disadvantage of pull, though, is that probing requests are periodic, generating a non-negligible *steady state load* proportional to the probing frequency. However, lowering the probing frequency to save on traffic overhead increases the dissemination delay, leading to delicate tradeoffs.

These complementary patterns of push and pull are the driving force behind the PULP protocol, described in the following section.

5.3 The PULP Protocol

From the observations presented above, it appears clearly that there is no ideal protocol performing well on all fronts. In this section we present PULP, a hybrid protocol harnessing the specific strengths of both approaches.

5.3.1 System Model

We consider a large set of n nodes communicating over an unreliable, fully connected medium (e.g., UDP over the Internet). Nodes can join or leave the network at any time. Departures and crash failures are treated equivalently, that is, there is no *graceful leave* operation. Byzantine behavior is out of the scope of this paper (see, for instance, BAR Gossip [81] for a dissemination protocol dealing with byzantine nodes).

All operations are *fully decentralized*. That is, there is no central entity to control any function of the system. All message exchanges (both periodic and sporadic) between nodes are *asynchronous*. Note that the dynamic and unreliable nature of the network rules out protocols that depend on rigid structures or reliable communication channels, such as tree-based dissemination protocols using TCP communication.

Regarding the anticipated workload, we consider (1) a *sequence of messages* being disseminated rather than a single message, (2) generated at variable arbitrary rates, and (3) originating at multiple nodes. As we will see, point (1) is particularly important as message disseminations are leveraged to inform nodes of previous messages that might have been missed, which in turn helps nodes adjust their pulling frequency. The resulting protocol is adaptive and self-controlled based on the current message generation rate. Note that the model of a sequence of messages matches the nature of many common applications, such as microblogging, RSS feeds, etc.

5.3.2 Supporting Mechanisms

Like many epidemic protocols, PULP relies on communication between peers selected *uniformly at random*. To that end, we rely on the family of PEER SAMPLING SERVICE protocols [63], and specifically CYCLON [129], which provides each node with a regularly refreshed list of links to random other peers, in a fully decentralized manner and at negligible bandwidth cost.

To provide a high level sketch of CYCLON we omit certain details found in [129]. In a CYCLON overlay, each node maintains a (very short) partial *view* of the network, that is, a handful of links (IP addresses and ports) to other nodes. Periodically, yet asynchronously, each node contacts a peer from its view, and they exchange a few of their views' links. As a result, views are periodically refreshed with new links to random other peers of the overlay. When the right policies are followed (see [63] and [129] for details), this method has shown to produce overlays that strongly resemble random graphs, that is, at any given moment each node's view contains links to nodes selected uniformly at random from the whole network. Moreover, this process has shown to converge in a few dozen cycles irrespectively of the initial topology, and due to the self-healing nature of CYCLON the respective properties are retained even in the face of node churn[1]. CYCLON and most other PEER SAMPLING SERVICE protocols have negligible computational, memory, and bandwidth cost,[2] and have shown to operate with remarkable reliability and robustness in (even highly) dynamic conditions.

To elaborate on the feasibility of nodes to communicate with randomly selected peers, when a node is equipped (through CYCLON) with a few links to *randomly selected* other nodes, and it *randomly selects* one among them, it is equivalent to having selected one node *at random* from the whole overlay. Further, when the node's CYCLON view is changing over time, the node has essentially access to an endless stream of random peers to communicate with.

Note that peer sampling protocols are also able to cope with the characteristics of actual IP networks, in particular with respect to nodes' reachability (as the node lies behind a firewall or NAT). The authors of [71] propose an augmentation of the CYCLON protocol that also deals with NAT-traversal issues while maintaining the same randomness characteristics for the overlay.

Finally, nodes in PULP need to have a *rough* estimate of the network size. For that, we employ the *interval density algorithm* [75], which can be executed *locally* on top of CYCLON with negligible cost. The principle of this algorithm is based on the fact that the CYCLON view provides a continuous stream of randomly selected peers from the entire network. Estimation of the network size relies on the density of these peers over a chosen value space, and proceeds as follows. Each node applies a hashing function to the IP addresses of each peer it discovers through CYCLON, mapping them to values uniformly spread in a given value space. It keeps a set of recent peers' hashed values that are the closest to a particular value (e.g., its own hash value), and uses the span

[1] In our experiments CYCLON converged in no more than 20 "cyclon rounds", that is 100 sec, and remained converged thereafter even at experiments involving churn.

[2] In our experiments CYCLON traffic accounted for an average of 24 bytes/sec per node, as explained in Section 5.4.1.

of this set over the whole value space to infer an indication of the network size. More than one hashing function can be used for increased accuracy.

5.3.3 PULP: The Intuition

As explained extensively in Section 5.2, push-based and pull-based protocols operate with opposite patterns. Conveniently enough, these patterns are complementary with each other regarding their strengths and weaknesses.

Given the respective observations, PULP strives to meet the following objectives:

Limit push. Let push execute only for the very few initial steps, to avoid redundant message forwarding while ensuring sufficient startup diffusion of messages. Our experiments (Section 5.4) indicate that reaching 4% to 5% of the network is a good target for the push phase, with nearly no redundancy.

Reduce redundant pulls. Avoid probing to find out *whether* a message is missing. Probe only in an attempt to pull the message, when it is known to be missing.

Adapt the pull period. Periodic probing constitutes a limitation. Too short a period causes unnecessary probing message load, particularly in periods of low message rate. Too high a period renders the system unresponsive when messages come at high rate. PULP is designed to dynamically adjust the probing frequency of nodes to match the current message rate.

The key observation is that if messages are forwarded to nodes selected uniformly at random, every message reaches a different set of nodes that is not correlated to the sets of nodes reached by other messages. Although a node might miss a given message with significantly high probability, the probability of it missing *all* of k messages, diminishes exponentially with k. We exploit this property in our algorithm.

With respect to the first objective, reaching 4% to 5% of the network in the initial push phase requires to set the values of TTL and FANOUT accordingly.[1] The coverage

[1] We chose this value of 4% to 5% as they allow for a low latency dissemination with only very few duplicates. Using larger values do not reduce the delays further but significantly increase duplicate counts. This choice is experimentally justified in Section 5.4.2.

obtained depends on both of these parameters, as well as the size of the network, which is estimated by the interval density algorithm (see Section 5.3.2). A node with a size estimation N_{est} simply chooses, for messages it generates, the values of TTL and FANOUT such that

$$c_{\text{est}} = N_{\text{est}} / \sum_{i=1}^{TTL} Fanout^i$$

is as close as possible to the expected coverage. Either TTL or FANOUT is fixed (possibly based on allowed range of values or on the CYCLON view size for the FANOUT) and the other parameter is derived accordingly. Our current implementation fixes the FANOUT to 3 and computes the value of TTL. Initial push messages can be sent with different TTL values to approach more closely the required coverage. Note that duplicates can be ignored in the calculation, as the value of the expected coverage is indeed required to be low enough to actually *avoid* most duplicates.

Regarding the second and third objectives, PULP leverages the push phase to relieve the pull phase of excessive probing requests. Instead of having each individual node explicitly probe random other nodes at fixed intervals to discover *whether* it is missing any messages, forwarded messages carry information about which other messages are available, conveying this information as a by-product of the push component.

5.3.4 PULP: The Protocol

We now present a detailed description of the PULP algorithm, which combines the push and pull components for disseminating a sequence of messages in a collaborative and decentralized fashion.

Algorithm 1 shows the pseudo-code of the PULP protocol. Each peer P maintains a *history* of the messages it has recently received, denoted as H_P. It additionally maintains a *trading window*, denoted as T_P, containing the list of messages that are available to other nodes on request.

When a message is pushed to (or generated at) node P for the first time, P registers it in H_P and, if the TTL has not been reached yet, forwards it to FANOUT random other peers. We stress that obtaining the IP address of randomly selected peers is a trivial task thanks to CYCLON, as described in Section 5.3.2.

In forwarding a message to another peer Q, node P also forwards the IDs of messages in its trading window T_P. These are messages that P considers to be in the pull phase, a subset of messages in P's history H_P. The trading window plays a key role in the interaction between nodes, because it helps nodes avoid exchanging messages that

Algorithm 1: Pulp algorithm on node P

Variables
- H_P: History of (recently) received message IDs
- Δ_{pull}: Period of pull operations (initially 30s)
- *missing*: Set of message IDs known, but not yet received
- *prevMissingSize*: Size of *missing* at the end of last Δ_{adjust} period
- $prev_{useful}$: Number of *useful* pull replies during current Δ_{adjust} period
- $prev_{useless}$: Number of *useless* pull replies during current Δ_{adjust} period

(Δ_{adjust}, TTL and FANOUT are fixed protocol parameters)

// Invoked when a message is pushed to node P by node Q
function PUSH(*msg, hops, Q,* T_Q)
- *// Forward further if needed*
- **if** *msg* received for the first time **then**
  - **add** *msg* to H_P
  - **if** *hops > 0* **then**
    - **invoke** PUSH(*msg, hops-1, P,* T_P) on FANOUT random peers
- *// Messages will be pulled at the next pulling period*
- *missing* $\leftarrow$ *missing* $\cup \{m \in T_Q : m \notin H_P\} \setminus \{\text{msg}\}$

// Periodic pulling of missing elements
thread PERIODICPULL()
- **do every** Δ_{pull} **seconds**
  - *// Shuffling reduces the probability of receiving duplicates by pull*
  - **shuffle** *missing*
  - **invoke** PULL(*missing, P,* T_P) on a random node Q

// Invoked when a node Q requests a message from node P
function PULL(*requested, Q,* T_Q)
- $m \leftarrow 1^{\text{st}}$ element in *requested* order $\in T_P$, or $\perp$ if none
- **invoke** PULLREPLY(m, *P,* T_P) on Q

// Receive a reply to a pull request from node P
function PULLREPLY(*msg, Q,* T_Q)
- **if** $msg = \perp \vee m \in H_P$ **then**
  - $prev_{useless} \leftarrow prev_{useless} + 1$
- **else**
  - **add** *msg* to H_P
  - *missing* $\leftarrow$ *missing* $\cup \{m \in T_Q : m \notin H_P\} \setminus \{\text{msg}\}$
  - $prev_{useful} \leftarrow prev_{useful} + 1$

// Periodic adjustment of pulling period for node P
thread ADAPTFREQ()
- **do every** Δ_{adjust} **seconds**
  - **if** $|missing| >$ *prevMissingSize* **then**
    - $\Delta_{pull} \leftarrow \frac{\Delta_{adjust}}{|missing| - prevMissingSize + prev_{useful}}$
  - **else**
    - **if** $|missing| > 0 \wedge prev_{useless} \leq prev_{useful}$ **then**
      - $\Delta_{pull} \leftarrow \Delta_{pull} \times 0.9$
    - **else**
      - $\Delta_{pull} \leftarrow \Delta_{pull} \times 1.1$
  - $\Delta_{pull} \leftarrow \max(\Delta_{pull}, \Delta_{pull_{\min}})$
  - $\Delta_{pull} \leftarrow \min(\Delta_{pull}, \Delta_{pull_{\max}})$
  - $prev_{useless} \leftarrow 0$
  - $prev_{useful} \leftarrow 0$
  - *prevMissingSize* $\leftarrow |missing|$

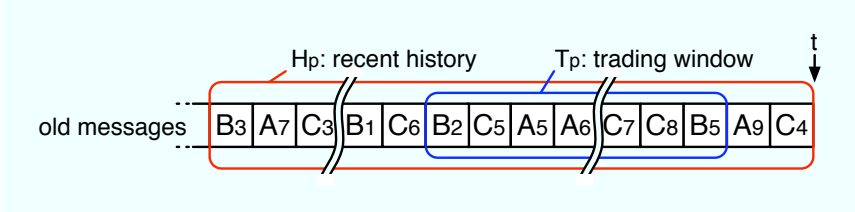


Figure 5.3 – Data structures of the PULP algorithm. Note that messages come from multiple sources (here A, B, and C) and each node sorts them based on the order it received them (which is generally different for each node).

are either (1) too recent and are still being pushed, or (2) too old and have already been removed from local histories. The trading window T_P essentially leaves a "safety margin" on both sides of history H_P. Figure 5.3 illustrates these data structures.

When Q receives T_P, it checks for messages not contained in its own history H_Q. If it discovers some messages it has missed, it inserts them in the *missing* set. These messages will be asked for by the periodical pull thread.

The periodic pull thread simply selects a random peer and sends it a pull request. The protocol does not try to pull from peers that are known to have the requested messages, neither does it keep information on which peer advertises what content. The rationale behind this design choice is that, while pulling from specific peers might slightly speed up dissemination of the first few messages, random selection has the advantage of distributing information about message availability more evenly (advertisements are sent along with pull requests), overall making the protocol more responsive and globally more efficient. It is a design decision that favors common benefit over (short term) individual gain.

As multiple pull operations of the same node may overlap in time, it makes sense to avoid requesting the same message from two different peers. Therefore, the set *missing* is rotated by one position (round-robin) before being sent next time. The inquired node selects, among the messages in its history, the *first* available one according to the ranking in the received *missing* list, if such an element exists. If none of the messages in *missing* is available, it replies with $\perp$, and the pull operation counts as useless. Note that, as for pushed messages, a node replying to a pull request also piggybacks its trading window in the answer.

The periodic adaptation of the pulling frequency is performed by a separate thread in the following way. Each node has a pulling frequency Δ_{pull}, and maintains a set of message identifiers, *missing*, that it has heard of but not received yet. Pulling

frequencies are chosen to follow the overall rate of new messages sent in the network. Every Δ_{adjust} seconds, a node inspects (1) the evolution of the size of the *missing* set during the last period, and (2) the number of useful and useless pulls that were performed during that period. If the size of the *missing* set has increased, Δ_{pull} is lowered to the period that would have been necessary to retrieve all newly known elements (assuming successful pull operations). That is, the new pulling frequency Δ_{pull} is simply the adjustment frequency Δ_{adjust} divided by the number of messages that should have been fetched during the last adjustment period to cope with a steady rate of reception (which would result in a steady size of the *missing* set).

This adaptation allows us to cope with increasing rates of new messages and to limit the growth of the *missing* set. If the size of *missing* has shrunk, the evolution depends on the ratio of useless vs. useful pull operations: if useless pulls dominate, Δ_{pull} is decreased by a small factor. If useful pulls dominate, Δ_{pull} is increased by the same factor. We observed based on preliminary evaluations that a value of 10% was yielding a good compromise between the reactivity and the accuracy of the pulling frequency adaptation mechanism. In other words, the pulling frequency aggressively adapts to sudden sending activity and adjusts to the highest possible value with acceptable useless pulling rate. Δ_{pull} can be bounded by $[\Delta_{\text{pull}_{\text{min}}}, \Delta_{\text{pull}_{\text{max}}}]$, depending on the underlying network properties and the desired reactivity of the system to new message sending activity.

5.4 Evaluation

This section describes experimental results from real deployments of PULP on PlanetLab [1], as well as a controlled deployment in a cluster. We first present our experimental setup and evaluation metrics. Then, we present experimental results demonstrating the performance, stability, and load in both static and dynamic environments. Finally, we compare PULP to a push-only and a pull-only protocol to highlight the benefit of its hybrid approach.

5.4.1 Experimental Setup

The implementation of PULP is based on SPLAY using UDP for communication. UDP, being a connectionless protocol, seems the most appropriate choice given the many short communication sessions between arbitrary nodes, rather than long sessions between fixed pairs. In addition, given the inherent fault tolerance of PULP, there is no

reason to opt for a more reliable (and expensive) protocol such as TCP. The ability of gossip-based dissemination to gracefully deal with packet loss is demonstrated in our experiments.

In all experiments, messages are 8KB in size and are sent from randomly chosen nodes of the network. Unless specified otherwise, the initial push phase is configured to reach between 4% and 5% of the network when there is no message loss (thus slightly less in practice). To that end, we set the parameters as follows: TTL=3 and FANOUT=3 in a 1,000 node network (ideally reaching 40 nodes—coverage 4%) and TTL=3 and FANOUT=2 in a 300 node network (ideally reaching 15 nodes—coverage 5%). We further justify this choice of a very low coverage of the initial push phase with our second experiment. Unless otherwise noted, the minimal and maximal pull periods are set to $\Delta_{\text{pull}_{\text{min}}} = 0.2$ seconds and $\Delta_{\text{pull}_{\text{max}}} = 30$ seconds.

No node has global knowledge of the network, and the selection of random peers for PULP operations is based exclusively on CYCLON [129], as explained in Section 5.3.2. CYCLON performs periodic pairwise shuffles of peers' views to maintain a constantly evolving overlay network whose properties are close to those of a random graph (i.e., each node's link is equally likely to be present in any other node's view). Views were configured to a size of 25 links to other nodes, and peers exchanged 5 links every 5 seconds. Given that a link is 6 bytes long (IP address and port), this accounts to 60 bytes of traffic (inbound and outbound) induced by each node every 5 seconds. Since this traffic affects two nodes, each node is involved on average in 120 bytes per 5 seconds, that is 24 bytes per second of total traffic for each node. Also, it should be noted that thanks to CYCLON's link aging policies, links to failed nodes can remain in other peers' views no more than 5 exchange rounds, that is, 25 seconds in our experimental settings.

We evaluate PULP along the metrics laid out in the introduction, namely coverage, dissemination delays, and redundancy (both in terms of redundant pushes and useless pulls). We evaluate (1) delays and their distribution, (2) the influence of the initial push phase, (3) the influence of high levels of churn on update reception delays and (4) the effectiveness of the self-adaptation of pulling periods for varying message generation rates.

5.4.2 Homogeneous Settings and Churn Resilience

Our first set of experiments was conducted on a local cluster composed of 11 dual-core nodes with 2 GB of memory each. Each machine hosts 91 instances of PULP, reaching a total of 1,001 nodes.

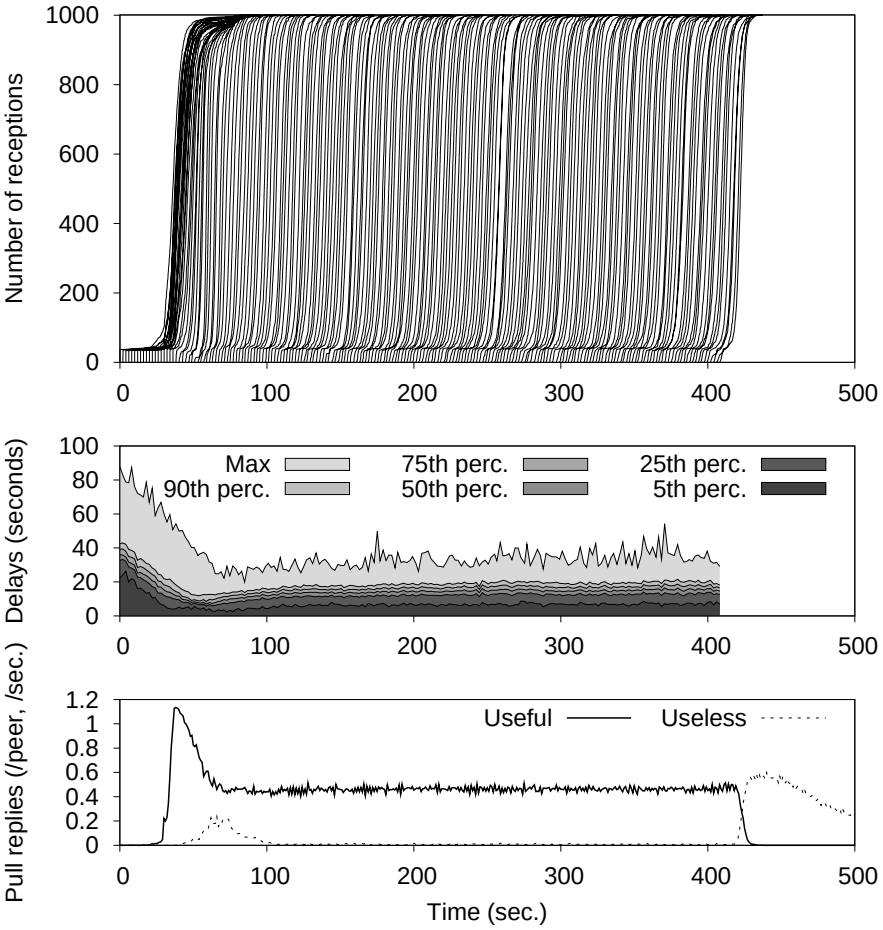


Figure 5.4 – Performance of the dissemination of 200 messages on a network of 1,001 nodes running on a cluster: individual cumulative delays, evaluation of the delay distribution, and evolution of the pull operations recall.

Our first experiment (Figure 5.4) presents the delivery delays for a stream of 200 messages. Messages are sent by randomly chosen peers at a rate of one every 2 seconds approximately.[1] The upper plot presents the number of nodes informed (through either push or pull) for each message with respect to time. Each single line corresponds to the evolution of a certain message. The middle plot presents the distribution of delays for each message. The abscissa corresponds to the time when the message is sent. For a given abscissa, the cumulative shaded areas represent the distribution of delays, by percentiles. For instance, the maximum delay for receiving the message sent at time 200 is 31 seconds, and half of the peers receive it within 14 seconds (50th percentile). One vertical set of percentiles (distribution) in the middle plot is a concise representation of the cumulative distribution of reception times for this message, shown by one individual line in the upper plot. Finally, the lower plot presents, for each period of one second, the mean number of useful and useless pull operations performed, per peer. This metric is used by nodes to self-tune the pulling algorithm, with the objective to reach a larger number of useful than useless pull operations.

We clearly see in Figure 5.4 (top) the initial push phase that reaches a small portion of the network (4%): the small vertical lines at the beginning of each diffusion, that is shown for all messages. The larger part of the dissemination takes place by pull operations. We observe an initial *bootstrap* phase, where mostly only push operations reach nodes, and no pulls take place. This is due to the fact that, prior to the dissemination of the first message, all nodes have pulling periods of $\Delta_{\text{pull}_{\text{max}}}$ =30 seconds in this experiment. As soon as enough nodes have been reached by initial pushes, there is a *warm up* phase with an increasing rate of periodic pull operations: nodes start to discover missed messages and to retrieve them. This is also demonstrated by Figure 5.4 (bottom): after the warm up phase, nodes start issuing pull operations, most of which are useful (pulling too early, on the opposite, would have resulted in a larger set of useless communications). As a small number of useless pull operations appear, the pull period gracefully adapts to the message emission frequency and only a very small fraction of pull operations are useless. As a result, the delivery delays for messages sent after the warm up phase remain stable.

The next experiment explores the impact of the initial push phase on the overall delivery latency: does a larger coverage during the push phase result in lower delays? Figure 5.5 (top) presents the distribution of delays for a set of 200 messages in the same settings as for the previous experiment. We vary the coverage by setting TTL=2 and by varying FANOUT from 2 to 16. The lower part of the figure presents the evolution of the proportion of duplicate messages as a function of the coverage of the initial push. For instance, we have a coverage of 14.4% of the nodes on average with FANOUT=12 and

[1] Due to the high load on our cluster, periods are not exactly respected by the machine's scheduler. This explains that the last message is sent at around 408 seconds and not 400 as expected.

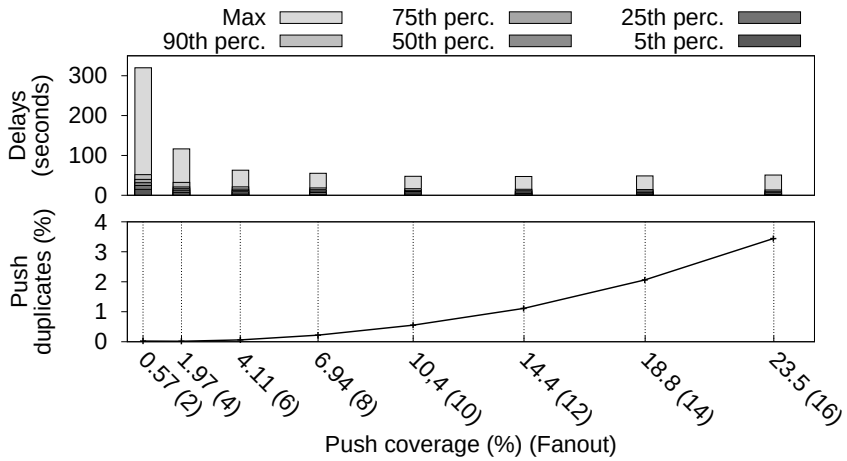


Figure 5.5 – Evolution of the reception latency distribution w.r.t. the coverage of the initial push phase.

this results in 1.11% of the nodes receiving at least one duplicate.

We observe the typical exponential growth of duplicates with respect to the coverage of the push phase: the higher the FANOUT, the more peers are reached, but redundancy grows faster than coverage. Most importantly, the delays that are observed with increasing coverage are decreasing only at very low values of FANOUT, and the price in redundant push operations overwhelms the benefits of higher values. This justifies the value of approximately 4% chosen as target coverage for our experiments.

5.4.3 Performance under Churn

The next experiment studies the resilience of the PULP protocol to churn (Figure 5.6). To that end, we use the SPLAY *churn manager* that can emulate node departures and arrivals in real time, by remotely starting and killing our prototype nodes at specific times. We replay a real trace of 2,000 nodes collected in the Overnet file-sharing network in 2004 [23] at its original speed, as well as 5, 10, and 20 times faster. The most accelerated run result in churn rates of 192 departures or joins per minute on average for an average population of 650 simultaneously active peers.

In addition to the nodes of the trace, we use a set of 100 static nodes, denoted as *observers*, to monitor the dissemination and reception of messages. A set of 200 messages is sent by nodes belonging to the non-observer set, one every 2 seconds. We plot

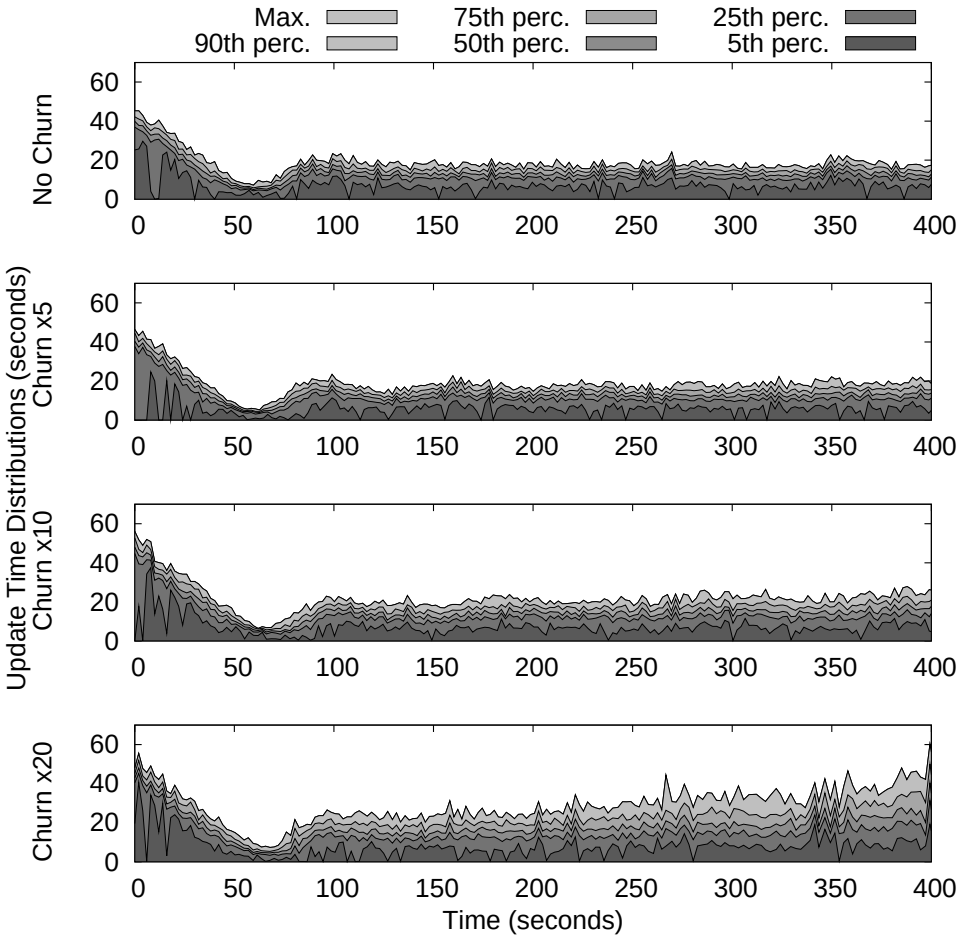


Figure 5.6 – Evolution of the delays as seen by a set of 100 *observers* (static nodes) under increasing churn rates.

the evolution of delays at the observers, for increasing churn rates, and for a static environment (with 650 nodes) as a baseline. As already mentioned, communication is carried over UDP, without ACKs being sent. Failed nodes are gradually removed by CYCLON only. This means that most nodes will have (temporarily) failed peers in their views, which results in additional message losses.

Figure 5.6 presents the evolution of the delays and illustrates the inherent capacity of PULP, to cope with dynamic environments. We observe that only a very high churn rate leads to slightly increased delays (lower plot of Figure 5.6). Interestingly, nodes pull more often when there is more churn because the period Δ_{pull} decreases as more messages are lost but the size of the *missing* set does not diminish. This demonstrates that the self-adaptation of the pulling period Δ_{pull} also deals with increasing loss in the communication and keeps the rate of reception sufficiently high for all online nodes to receive all published messages.

5.4.4 PlanetLab Experiments

Our second set of experiments is run on the PlanetLab world-scale distributed testbed. PlanetLab is composed of nodes that are extremely heterogeneous in load and available network resources. We use a set of 300 randomly chosen PlanetLab nodes. We evaluate dissemination delays and analyze the self-tuning of the pulling period with varying message emission frequencies.

The experiments carried out on the Planetlab testbed highlight the ability of PULP to achieve full coverage in heterogeneous environments that are prone to failures, message loss, and arbitrary delays. As a matter of fact, Planetlab nodes experience significantly less reliable IP communication than typical computers on the Internet, due to their massively parallel virtualization and high load.

The first experiment reproduces the scenario studied in the cluster and shown in Figure 5.4, with the notable difference that we use here 300 *distinct PlanetLab nodes*. The data representation is the same as for Figure 5.4. A set of 200 messages are sent by nodes selected randomly at a rate of one message every 3 seconds (again, the time span deviation from 600 to 730 seconds is due to scheduling issues on heavily loaded nodes).

One can observe in Figure 5.7 that, as before, the first messages help to *bootstrap* the dissemination process by notifying nodes of some publishing activity. Messages then need approximately 30 seconds to reach half of the network and all nodes but the 10% slowest ones receive them in less than 60 seconds on average. Note that some of the randomly selected PlanetLab nodes failed or became unresponsive during our

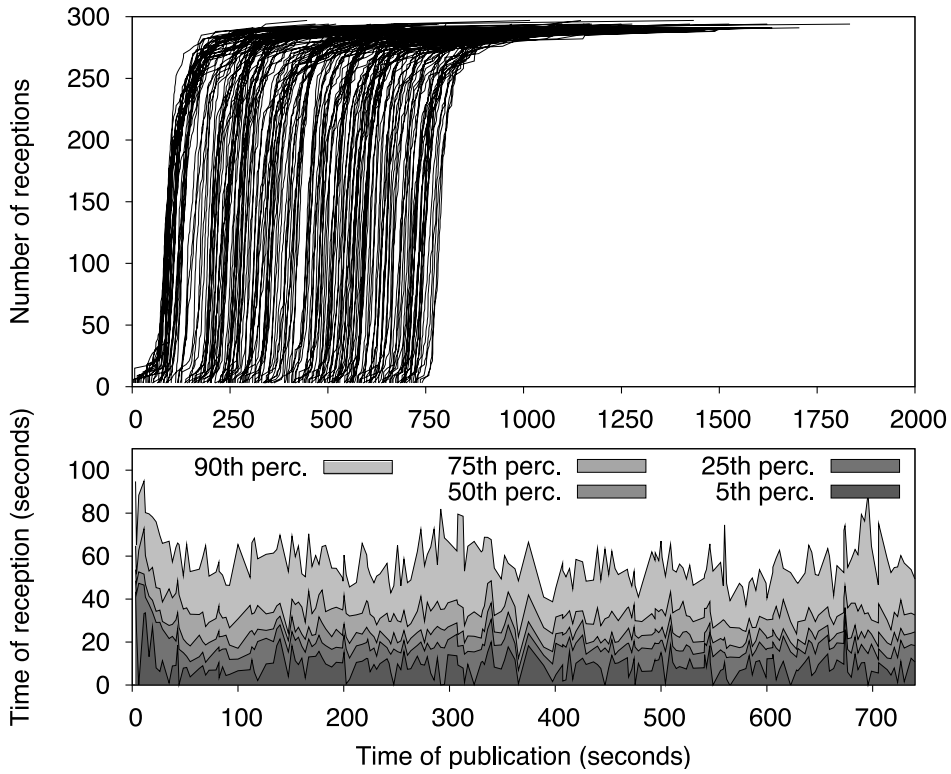


Figure 5.7 – Performance of the dissemination of 200 messages on PlanetLab.

experiments, as one can observe in the figure: the protocol achieves a coverage of 100% of all live nodes at slightly less than 300 receptions.

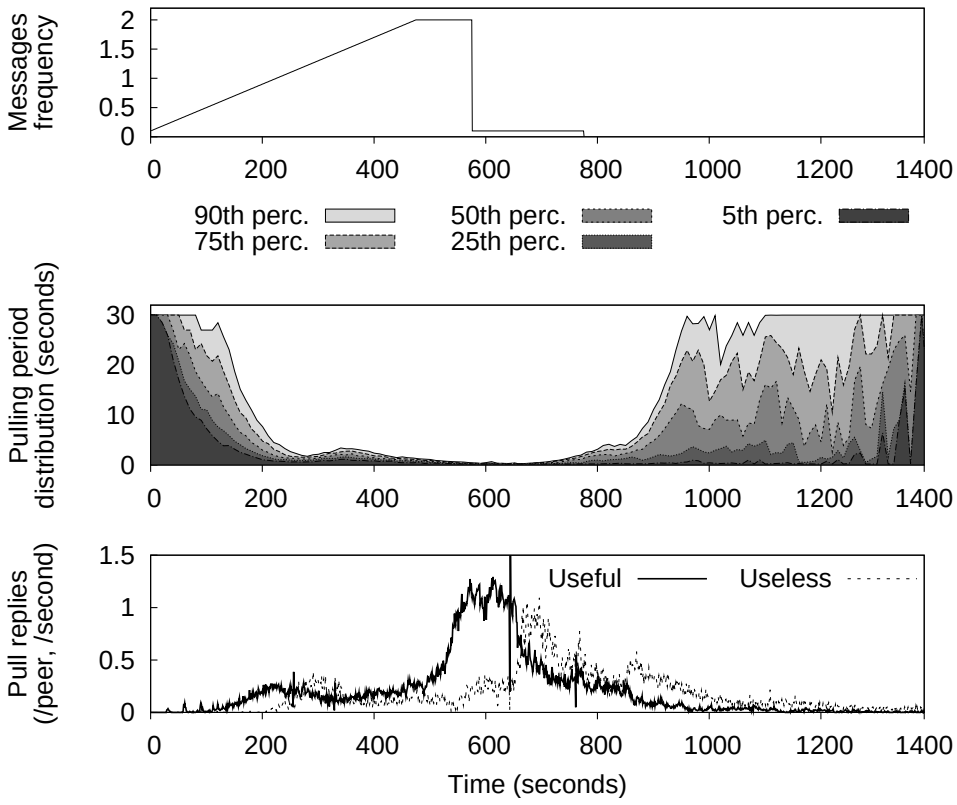


Figure 5.8 – Evolution of the pulling frequency w.r.t. new messages frequency.

Our second PlanetLab experiment evaluates the capacity of PULP to self-tune the pull period Δ_{pull} on each peer. We consider again a network of 300 PlanetLab nodes with only one publisher, whose message sending rate follows the frequency evolution of the upper plot of Figure 5.8: starting from a frequency of 0.2 (i.e., one message every 5 seconds) and increasing to 2 messages per second, for a duration of 475 seconds (100 messages). This steady increase is followed by the sending of 200 additional messages at a rate of 2 messages per second, followed by a sudden drop to one message every 5 seconds for the last 20 messages. 320 messages are sent in total.

We observe in the middle plot the distribution of the pulling periods Δ_{pull} computed by the self-adaptation algorithm: the initial frequency is of one pull every 30 seconds (*idle* state). As soon as messages are published, nodes discover that they miss some newly published messages and adapt their frequencies accordingly. The lower plot presents the number of pull requests issued by nodes, distinguished between useful

(i.e., resulting in the reception of new information) and useless requests (i.e., retrieving no new message). We observe that the pulling frequency follows the evolution of the new messages frequency in the first phase, with a slightly oscillating behavior around the optimal value. This oscillation is typical of the control-loop-based adaptation of the pulling frequency implemented by Pulp, and is also a result of the latency between the observation and the adaptation (as adaptation decision are made periodically, on average half such a period is required before the adaptation takes place). This does not result in significant delay increases for individual messages. Then, as messages are being disseminated, most pull operations are useful and the pulling frequency remains high. As soon as most of the messages have reached all nodes, the number of useless pulls grows, resulting in an increase of the pulling period.

Overall, we observe that most of the pulls are successful and result in delays low enough to sustain the message sending rate during dissemination, whereas a pull-based protocol with a fixed pulling period would have either incurred high delays or a high number of useless pulls. This observation further highlights the importance of the self-adaptive pulling period.

Our final experiment evaluates the capacity of Pulp to react to bursty message generation scenarios. It complements the previous experiment (Figure 5.8). Figure 5.9 presents the dissemination of a total of approximately 2,000 messages in a network of 200 nodes. To better observe the behavior of the dissemination after the burst, we voluntarily slow down the system reaction by setting a very conservative minimum period for pull replies of one second. The maximal period is maintained at 30 seconds. Note that the dissemination speed in fully-loaded mode (i.e., when many messages are on-the-fly in the network and are still being propagated) is a direct function of this parameter, which in turn proportionally impacts the dissemination time.

Messages are sent from all nodes in the following manner: an initial message is sent by a first node, and other nodes react to the first message reception by triggering between 1 to 20 messages (their number is chosen randomly). This scenario illustrates the behavior of Pulp when messages are sent as bursts by combinatorial reaction. We observe on the upper plot the number of messages sent per second (from any node in the network). Note the log scale on the y axis. We start from a steady state system with all pulling periods equal to 0, and one message is sent initially. Nodes receiving the message by the initial push phase generate new messages which result in 198 messages being sent after a warming period of 180 seconds, and a peak of 783 messages once all nodes receive the initial trigger message as they turn to the minimal pulling frequency allowed of one pull request per second.

As there are messages remaining from the burst to be received by nodes in the network, nodes continue to pull with a near-optimal success rate (the average number of useful

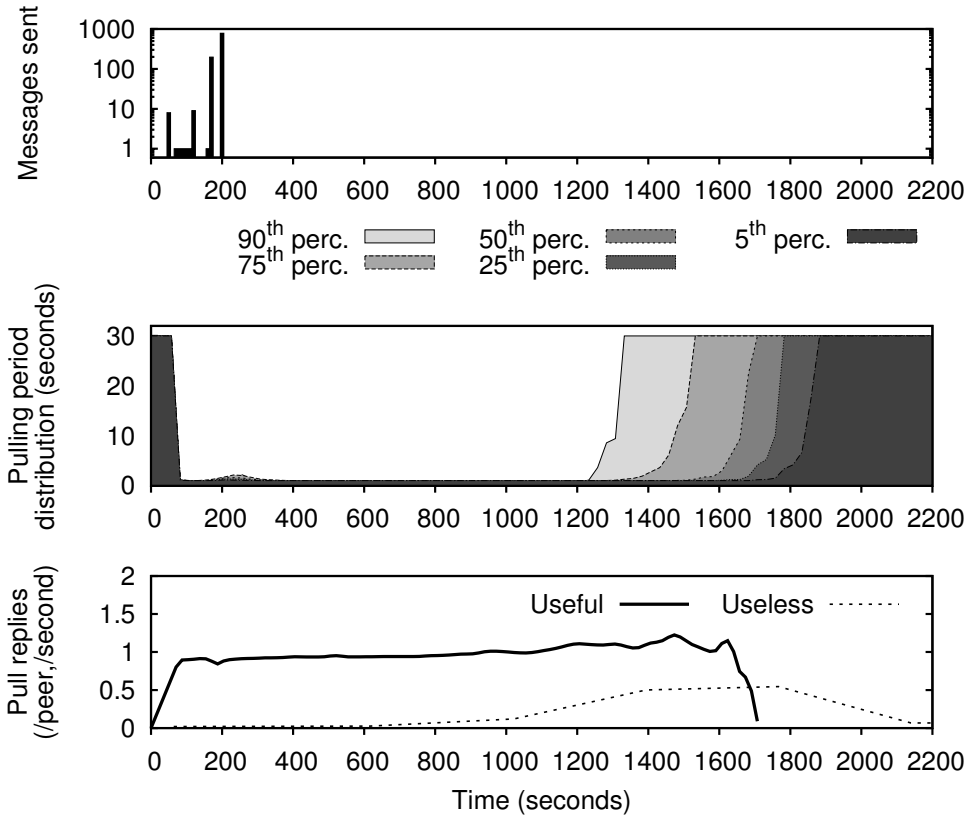


Figure 5.9 – Reaction to a message burst with a one second minimal pulling frequency.

and useless pull requests are shown on the bottom graph). We observe that nodes stop pulling approximately at the same time. This is an expected result as all nodes are pulling at the same (minimal) period and the large number of messages to be received is much larger than the number of nodes that are reached by the initial push phases. Moreover, these initial push phases reach different set of nodes, allowing all nodes to warm up their pulling activity while distributing initial messages in similar amounts to all of them.

This experiment highlights the ability of Pulp to disseminate at a very low cost (one message per second and per node), large number of messages arriving as burst from multiple senders. We note however that, as claimed in our initial assumptions, Pulp is not tailored nor designed to handle reactive systems (such as eventing mechanisms) where the delays of reception of particular messages are critical. Instead, Pulp carries its goal of disseminating in an adaptive and robust manner sets of messages from multiple sources at extremely low cost.

5.4.5 Comparison to Push-only and Pull-only Disseminations

Our last experiment highlights the benefit of using the Pulp hybrid pull-push approach to dissemination, as opposed to a solution that would use only push or only pull operations. We compare Pulp against two protocols:

- A "pull-only" adaptive protocol. This is basically a subset of Pulp, where the network is not seeded by some initial push phase, but where we keep all other features, in particular, the pulling period dynamic adaptation.

- A "push-only" algorithm, which works as follows. A node that wishes to publish some message initiates a push phase, as for the regular Pulp protocol, but this operation is not completed by regular pull operations. The push itself does not intend to reach a full coverage of the network, which would be totally impractical and overflow any routing infrastructure quickly. Instead, subsequent push operations are used to convey *implicit pull offers* about ongoing disseminations and publicize the availability of new data. When a node n_a pushes some new message m to some node n_b, n_a includes its list of message identifiers (in the same way as in normal Pulp). This list is sorted in the order of message reception as seen by n_a (as all nodes can publish messages, there is no global order on messages, thus two nodes can have different orders for the same set of received messages). In return, n_b requests from n_a the *first* message from n_a's list (the oldest as seen by n_a), among the ones it does not already have. In this way, older messages get larger priorities and, as there are more messages to disseminate, the number of implicit pull offers grows accordingly and helps resolve previous messages.

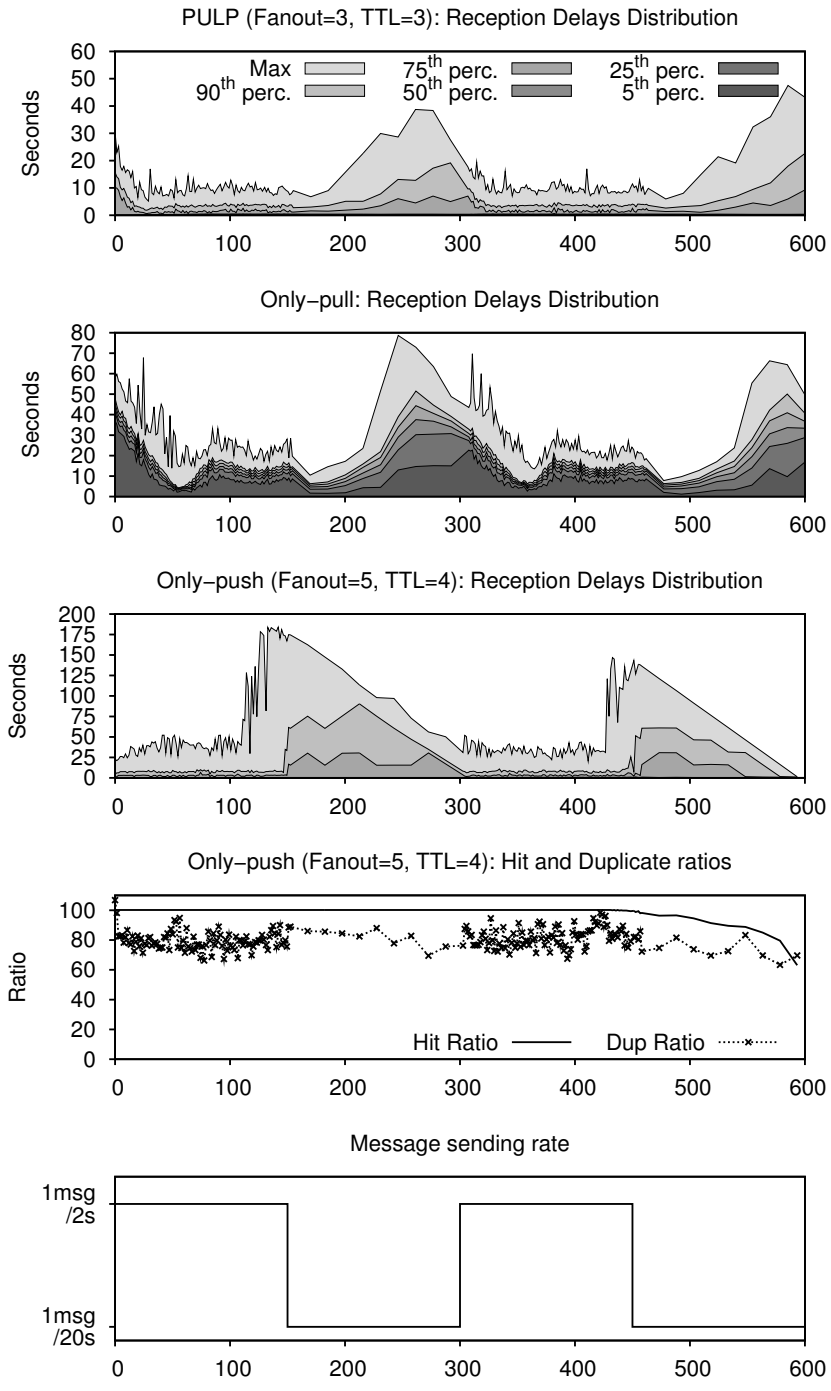


Figure 5.10 – Comparison of Pulp with an initial seeding phase (top), with no initial seeding phase hence relying only on push operations (middle) and a dissemination that uses only push to disseminate new messages and implicit pull proposals.

These two protocols help to highlight the behavior of PULP itself, as both implement its two key ideas: (1) using pull frequency adaptation to adjust to changes in message emission rates and (2) leveraging a sequence of disseminations from various sources and the spread of information about which elements are missing by an initial push phase.

Figure 5.10 presents the distribution of dissemination times for a set of 200 messages on a cluster hosting 500 nodes for the three protocols, as well as the duplicate and hit rates of the push-only protocol. Messages are published from random nodes in the network, but according to the frequency that is shown by the bottom plot: alternation between a frequency of 1 messages every 2 seconds and 1 message every 20 seconds, for periods of 150 seconds.

The top plot presents the dissemination times obtained by using PULP, which are consistent with previous evaluations. Periods of high message sending frequency imply low delays as the number of transient messages in the network is kept to a minimum, yielding more pull messages to sustain the frequency of new messages. Periods of lower sending rate result in lower activity hence higher maximal reception delays, but remaining at acceptable levels with a median delay of at most 10 seconds and a maximal delay of 30 seconds (note that this can be reduced by setting $\Delta_{\text{pull}_{\max}}$ to a lower value).

The second plot presents the dissemination delay for PULP without the initial push phase. These results convey those from Figure 5.5: delays increase drastically when the *seeding* effect of the initial push phase is absent. Median delays are effectively doubled when compared to the classical PULP.

The third and fourth plots present the dissemination results for the pure-push scenario: delays distribution, hit ratio (proportion of nodes receiving a message) and duplicate ratio (proportion of useless message reception compared to useful ones). The hit and duplicate ratios are not shown for PULP and its variant with no push phase as the former is always 1 and the latter always 0 by construction. We set the values of TTL and FANOUT for the push-only solution as the minimal ones that yield a hit ratio of 1 for most messages. Under such conditions, the duplicate ratio (mostly due to duplicates during the push phase) is around 80%, that is, a message is received on average 900 times in a network of 500 nodes. The lack of regular pull messages has the consequence that messages are not comprehensively disseminated when no further dissemination takes place, or when the message sending rate drops suddenly.

Overall, this experiment shows that both key components of PULP are necessary for its proper operation, and that, accordingly to the intuition, a hybrid and adaptive approach yields the best results.

5.5 Related Work

Unlike structured dissemination overlays (typically using trees) that use reactive strategies to tolerate failures and churn, gossip-based approaches proactively implement redundancy in the system and trade network overhead for extreme robustness against failures. Building upon the seminal work in this area where epidemics were introduce to disseminate updates in a database [45], many probabilistic gossip-based approaches have been recently proposed [26, 48, 55, 84, 85]. There has been a growing interest in such proactive approaches in a context where large-scale systems are highly dynamic. They avoid the need for potentially slow recovery mechanisms as well as the cost of maintaining structures in overlay networks. More specifically, many approaches have been proposed in the area of gossip-based video-streaming and secure protocols, breaking the myth of the lack of relevance of gossip protocols to distribute bandwidth-intensive contents or cope with malicious behavior [28, 80, 81, 82, 103, 101].

Gossip-based dissemination protocols usually fall under either of the push-only and pull-only category of protocols as discussed earlier, both having their own tradeoffs with respect to overhead (message redundancy), delay and robustness.

The aforementioned collaborative video streaming protocols combine the two methods in a context-dependent manner: typically push-only protocols are used to disseminate control messages so as for peers to subsequently pull useful stream packets. While the two protocols are combined, they are used for different purposes precisely because they have different characteristics. Push protocols are robust but introduce redundancy and are relevant to disseminate control messages where robustness is required and messages are typically small. Pull protocols are used for the dissemination of large content to limit redundancy. Chainsaw [101] uses pull only to pull for new data in a dynamic network based on a peer sampling mechanism. BAR Gossip [81] and Flightpath [82] focus on tolerating the presence of byzantine peers. In Coolstreaming [80], the content location is pushed while the actual content is pulled as in swarming systems. HEAP [54] accounts for peers heterogeneity by letting nodes dynamically adjust their contribution to gossip dissemination according to their capabilities. In [28], several push-only gossip-based approaches that differ in the choice of the content being pushed are studied. More specifically, the authors demonstrate that sending the most recent chunks to random peers achieves close to optimal dissemination with respect to rate and delay. In [36], a push-only protocol is combined with fountain codes (rateless erasure-correcting codes) to eliminate the unecessary redundancy of standard push protocols.

The approach of Pulp is to combine push and pull, not simply to disseminate control messages on one hand and the actual content on the other hand. This approach is shared to some extent by the following work.

The Interleave protocol [114], which is further evaluated by [41], combines push and pull for set of messages as PULP does, but with an approach that differs in several aspects. In Interleave, a list of sequential items is considered. Push is used to propagate new items to a large set of nodes in the system, while pull is used to retrieve the oldest missing items from the set (as decided from the sequence number of newest messages received). As such, Interleave focuses on single-publisher scenarios while PULP targets multiple publishers. Moreover, the frequency of pull operations does not adapt to the frequency of new messages, incurring either a potentially high steady-state load or a lack of reactiveness in periods of heavy publishing activity. On the other hand, an interesting contribution of Interleave is to take into account the bandwidth limitations at each peer, thus supporting more easily high-bandwidth file diffusion in heterogeneous systems. The properties of push/pull protocols for live video streaming are further studied in [113], with the interesting conclusion that RTT is an essential parameter for such delay sensitive systems (different from those considered by PULP).

In [88], the authors propose a hybrid dissemination mechanism that also aims at reducing the cost of the push phase by limiting the number of duplicates, and relying on a pull phase thereafter to complete the dissemination. Nonetheless, the main difference with PULP lies in the approach that is used for limiting the duplicates in that push phase. In [88], a structured network based on prefix routing is used conjunctively with a random network. Both the structured and random overlays are constructed in a delay-aware manner. The structured network is based on a coarse-grain structure, using only a few digits for prefix based routing, allowing a more robust construction and maintenance. The push phase is based on prefix-precedence relations and uses embedded trees that are found in the structured network to *seed* the network with the new messages. Meanwhile, the approach does not allow for frequency-adaptive pull operations. Interestingly, the authors of [88], based on simulation of their protocol, share our observation from Section 5.4.2, that reaching a small fraction of peers in the initial push phase has only a small increased delay when compared to a push phase that seeds most of the network. This observation pledges in favor of the PULP approach that stops pushing messages before duplicates are likely to occur rather than maintaining a structure amongst peers to ensure this property, as the former is more lightweight and robust in the long term. In any case, the two protocols focus on optimizing different metrics: [88] concentrates on minimizing delays, while PULP prioritizes on lowering traffic, be it redundant data transfers or control messages.

In [66], Karp *et al.* theoretically analyze the combination of push and pull for disseminating a *single* message. They propose using push-based dissemination until $n/\log n$ nodes have the message with high probability (*exponential growth* phase). From that point on, each uninformed node has sufficiently high probability of reaching an informed node by probing nodes at random, so they switch to pull communication (*quadratic*

shrinking phase). The overall message complexity is $\mathcal{O}(n \log \log n)$. As the algorithm relies on an exact estimation of the number of rounds to execute during the first phase, Karp *et al.* propose a *median-counter* algorithm to determine the right phase transition time. Nonetheless, contrary to PULP that strives to avoid duplicate message deliveries, the goal of this mechanism is to reduce delays. Hence, the first phase (push) is used for as long as the probability of hitting a non-informed node is higher than the probability of a non-informed node randomly probing an informed one. As a side-effect of limiting the push phase, the number of redundant deliveries is reduced to some extent, however, it is far from being eliminated. Moreover, Karp *et al.* focus on the independent dissemination of *individual messages*, not taking advantage of streams of messages and the potential interplay these messages can have in enabling faster or more reliable dissemination overall.

Liu *et al.* [87] also use a hybrid mechanism that mixes push and pull interactions for cache replica maintenance. In this context, a push operation refers to a proactive replication from the "master" node and a pull operation to a passive replication from nodes holding replicas to nodes with free space as a result of periodic exchanges.

Also in the domain of cache updates, Srinivasan *et al.* [120] and Urgaonkar *et al.* [126] proposed to dynamically adapt the frequency of pull requests from replica holders to master nodes for the dynamic update of read-only replicas in caches. Adaptation is performed in a similar manner as in PULP, by adding a constant to the frequency as the number of updates to replicas increases, and dividing the frequency when experiencing too many useless pull requests.

5.6 Summary

The properties of gossip-based protocols have been widely studied in the literature. Such protocols rely on randomization and are known to be simple, scalable and extremely robust. Yet, they have long been deemed impractical, mainly because of the large gap between their behaviors as predicted by theoretical models and as experienced in real networks. More specifically, the correct operation of gossip-based protocols depends on many external factors that are difficult to dimension properly without good knowledge of the underlying system (e.g., its size, delays, lossiness, etc.). Moreover, gossip-based dissemination protocols are usually considered much more expensive than deterministic protocols in terms of overhead. Yet, the robustness to churn of such protocols make them more and more appealing to the point that they have recently been used in the context of video streaming applications.

In this context, we have presented Pulp, a lightweight gossip-based dissemination protocol that combines pull- and push-based approaches and performs remarkably well in practice. Pulp disseminates flows of messages originating from multiple sources to large sets of nodes in a fully decentralized way. Gossip messages exchanged by the push protocol carry information that helps nodes perform pulls in a smart and self-adaptive manner.

Thanks to its hybrid approach, Pulp limits the redundant traffic from the push phase and the unnecessary polling from the pull phase, thereby being particularly network-efficient. Our deployment of Pulp in real-world conditions on a cluster and on PlanetLab demonstrates its good performance and its robustness even in the face of high churn. While Pulp seamlessly supports node failures, it does not explicitly take into account the presence of selfish or malicious nodes. This is an interesting area of research left for future work.

Chapter 6

Collaborative Ranking and Profiling

> Everything should be made as simple as possible, but not simpler.
>
> A. Einstein

6.1 Introduction

Search engines certainly play the most significant role in today's Web usage. Leading search engines rely on the observation of the structure of linked elements [32] (i.e., the graph formed by hyperlinks between pages and data items), which is used in conjunction with the keywords forming a query to decide on the most relevant elements, or for advanced approaches with user-centric search options and hints (e.g., when using Google's SearchWiki [13]). These search engines do not leverage the *collective knowledge* that is created by the users as part of their navigation choices. Instead, the bulk of the score used to decide on this relevance depends on the links pointing to the element, that is, scores are mostly based on *structural* information. While efficient for retrieving the most relevant elements in case the implicit semantic search area (i.e., *interest domain*) is the most popular one, there exist many situations where the elements that are the most cited ones, or belong to the most renowned sites are not those expected by the user.

For instance, a Web search for the query term "Java" returns a list of elements mostly focusing on the programming language. This is obviously a result of the predominance of computers-related resources on the Web. Nonetheless, a user looking for information on the Indonesian island of "Java" will be dissatisfied by not finding any relevant information (from her point of view) before the items of rank 6 and 16.[1] The solution for avoiding such a situation and obtaining better-tailored results is to pair the structural information used by the search engine with some *semantic* information about the expectations of a particular user. Concretely, information about which items were deemed interesting by other users with similar interests can be leveraged to avoid search domain inadequacies. As a result, the *information diversity*, which is not well captured by solely monitoring the structure of the information graph, can be achieved by taking into account the diversity of expectations from querying users and using the *wisdom of crowds*, learned from past accesses, to determine relevant content for one particular user.

Information about one user's interest can be derived from the set of elements that she accessed as a results of her previous queries (*feedback information*), and from the keywords forming these past queries themselves. Similarly, the set of elements that are deemed interesting by users of some semantic interest profile can be derived from the elements they accessed after a Web search, that is, relevant elements can be extracted by correlating user accesses and extracted interests.

We believe that the best approach for proposing such a service is to build a *companion service* to complement search engines, instead of creating a new stand-alone search mechanism. Indeed, despite many research efforts invested so far to propose collaborative search engines (*e.g.*, Faroo, YaCy, Wowd[2]), no system has been able to reach a sufficient level of quality and efficiency to truly compete with its centralized counterparts. This is a direct consequence of the *bootstrap problem* [58]: added value of a new collaborative search engine becomes perceivable only when the system has attracted enough users to fully sustain its specific functionalities.

Figure 6.1 presents a general vision of the companion service: the user sends her request to a keyword-based search engine, which returns results based on structural information.

Meanwhile, the same query is sent to the collaboratively built companion semantic search service. Note that the latter request is paired with some semantic profile, which is a representation of the user's interest field. The companion service then returns a set of elements tailored to the user requirements on the basis of her semantic profile, and

[1] On http://www.google.com at the time of writing.
[2] http://www.faroo.com, http://YaCy.net, http://www.wowd.com/.

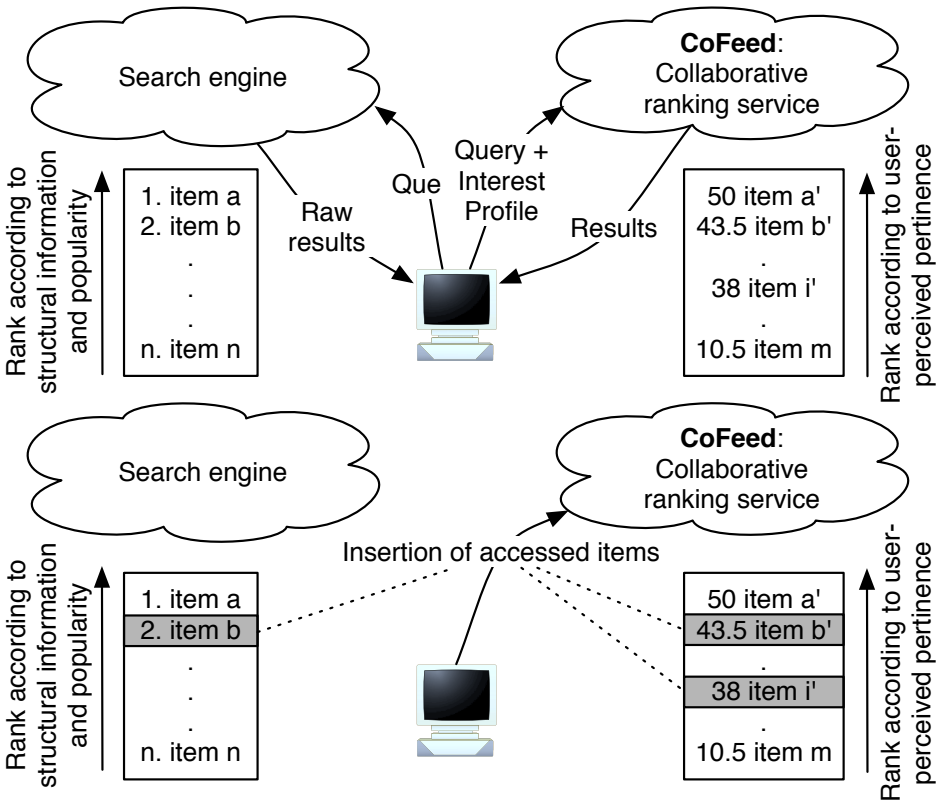


Figure 6.1 – Usage of a companion search service.

ranked according to their relevance to her interest domains. The additional results can then be presented together with the results from the traditional search engine used, in a similar manner that context-sensitive ads are presented as suggestions to the user for a query on most current centralized search engines' results. This simple presentation is also used by [99]. Although more elaborate presentations of the results to the user can be devised, we consider this to be a research task on its own, and not the focus of this chapter. Information about subsequent accesses (i.e., which item is accessed for some query and in which order) are sent to the semantic ranking service and used for building, for each request, a set of items that preserves information diversity.

Building such a system poses a set of challenging research issues related to information management (Section 6.3). First, how to accurately capture the semantic information associated with user activities (profiling interests, using actual accesses to construct a representation of some user's interests)? Second, how to process the feedback information to maintain sets of relevant elements that capture information diversity? Third, how to efficiently construct from these sets a tailored ranked list of results to answer user requests?

Another challenging question, which this chapter answers in detail (Section 6.4), concerns an appropriate infrastructure for supporting such a service. A centralized approach is easy to implement but scalability in number of users comes at a prohibitively high cost, especially if the service also has to tolerate failures. Moreover, it poses again a *bootstrap problem*, with many resources necessary before being able to serve a reasonably sized set of users. On the other hand, a distributed (*collaborative*) architecture has a much lower cost of bootstrap, and as the number of users increases, the number of servers also increases.[1] Last, beside relieving the bootstrap and scalability issues, distributed architectures are known to be better candidates for implementing fault tolerance and for balancing the load of serving clients over a large set of collaborative machines.

The fault tolerant architecture has been built using SPLAY. We reused the code made when implementing Pastry [110] to provide the distributed DHT and the routing protocol. To update the *Bloom filters* [27] necessary to store users preferences, we needed efficient binary operations. For that reason we added a new library in the SPLAY framework whose processing part is directly implemented in C.

The remaining of this chapter is organized as follows. We start by giving a general overview of the system in Section 6.2. We present the ranking and profiling mechanisms

[1] Note that end users are not necessarily acting as servers as in a *pure* peer-to-peer model. Instead, institutions can dedicate one or a few servers for provisioning the system as the popularity of the service increases—hence the collaborative aspect.

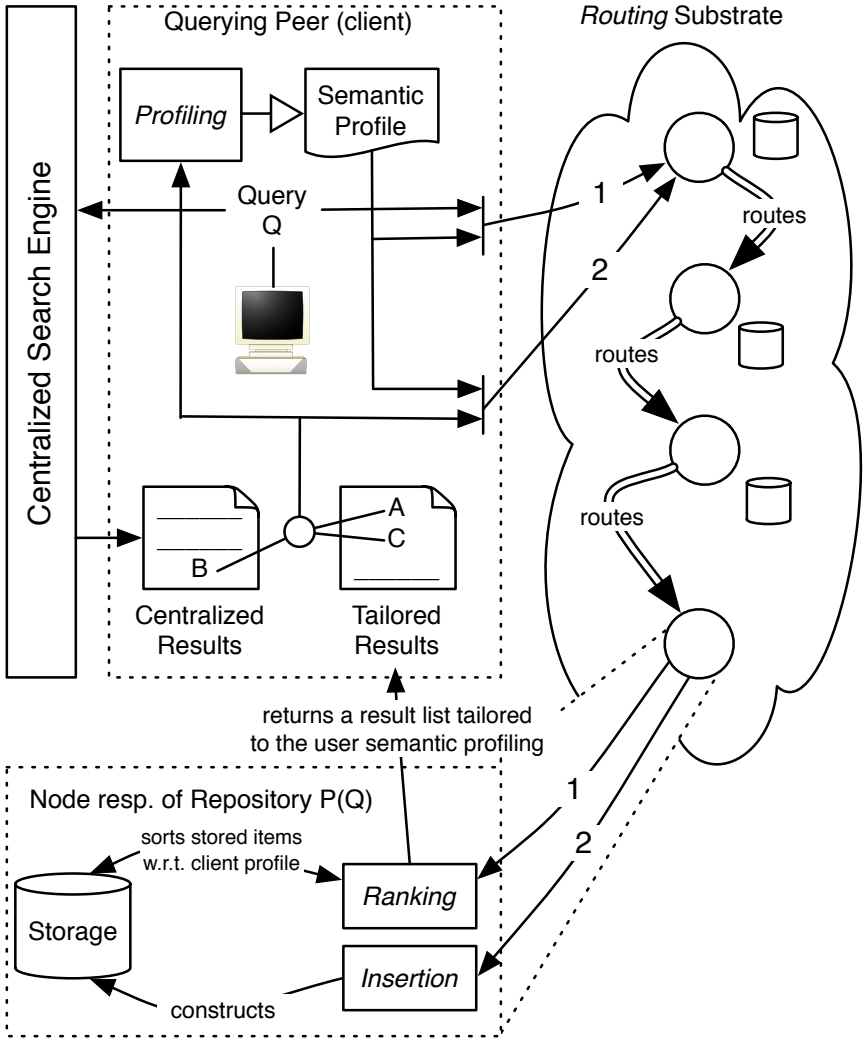


Figure 6.2 – Components & information flow.

in Section 6.3. The distributed supporting infrastructure is presented in Sections 6.4. We present the evaluation of all mechanisms in Section 6.5. We present related work in Section 6.6 and some perspectives in Section 6.7, before concluding in Section 6.8.

6.2 Overall Architecture

Before describing in detail the components and algorithms of CoFeed, we start by a general overview of its architecture, as depicted in Figure 6.2. CoFeed consists of a software component on the client computer (typically a browser plugin) and a distributed infrastructure that implements the collaborative ranking. The distributed infrastructure is composed of a possibly large number of nodes that collectively store and update repositories of items and associated relevance feedback information.

Queries from a client are sent to some existing search engine. At the same time, they are sent to CoFeed together with user-specific *interest profile* information. The *routing* substrate is in charge of delivering the query and the profile to the appropriate node, responsible for the target query's repository (see arrows labeled 1 in Figure 6.2). Based on the query terms and the profiling information, the *ranking* module on that node produces a ranked result list tailored for the user. The client can then combine the lists obtained from the search engine and CoFeed to improve the overall quality of the results presented to the user.

Relevance information is gathered on the user's machine by observing accesses to elements returned by any of the search methods (documents A, B, and C in the figure). This information is used by the *profiling* module to consolidate the local interest profile of the user. It is also sent to the *insertion* module on the node that is in charge of the query repository, and along with the profile of the user, to update the relevance tracking information for this query (see arrows labeled 2 in Figure 6.2).

6.3 Profiling, Storing and Ranking

This section describes how our system gathers profiling information, processes user queries, and stores and ranks relevant documents. The set of information associated to one query and stored in CoFeed is called a *repository*. We use the notations and terminologies given by Table 6.1.

6.3.1 Profiling user interests

The accesses of users to documents form the basis for constructing their local user interest profile (UP). Each document from the result list is associated with a snippet, which contains a larger set of keywords (or *tags*) representing the content of the document. These keywords are used to form the interest profile (UP) of the user, which is used in turn to construct document profiles (DP) maintained in the distributed repositories. Keywords are normalized in the system by classical means (stemming, noise-word list, alphabetical sort, duplicate words elimination). As storing all keywords for all accesses is obviously not possible, CoFeed represents profiles using *Bloom filters* [27], which are space-efficient probabilistic data structures allowing fast and false-negative-free inclusion tests over a set of elements. Moreover, Bloom filters have the additional advantage of increased privacy, as in addition to the identity of the user already being hidden, the keywords forming the UP are not available in clear text either to the server processing it.

A Bloom filter maps elements from an unbounded set to a bounded set of k bits in a bit array of medium size (8,192 bits in our prototype) by using k different uniform hash functions (we use 3 hash functions in CoFeed). Elements (keywords from snippets and queries) are inserted in the profiles (UP) by setting the k bits corresponding to these hash functions in the associated filter. The inclusion is tested by checking the bits corresponding to each of the k hash functions, and can yield some false positives. This is not much of a concern in CoFeed, as Bloom filters are not used for inclusion tests but for estimating union and intersection sizes of two sets. This is based on the number of bits set in the logical OR and the logical AND of the two filters. In CoFeed, we compare two profiles S_1 and S_2 by using the Jaccard similarity: $\frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}$. This similarity metric between an UP and a DP represents the *adequacy* to the user interest domain of a document. The same metric between two DPs represents their *semantic proximity*.

In order to avoid the saturation of bloom filters over time as new queries are performed and as more feedback is inserted in CoFeed, we use for both document and user profiles a variant of bloom filters called *time-decaying bloom filters* [40]. In this variant, bits that are set are associated to decaying timers. Newer elements have a higher weight and older information gradually disappears over time. The larger memory required for each bit is compensated by the frequent removal of elements (and thus the clearance of some bits) from the set. Using this structure allows CoFeed to spontaneously adapt to variations in the popularity of queries and users to receive a feedback that is more relevant to their ongoing search session.

Q	Query (a set of keywords, normalized by stemming, stop words removal, etc.)
$P(Q)$	Node in charge of the repository for query Q
$RFitem$	A relevance feedback item (composed of Q, D, UP, $Snippet$)
D	URL of a feedback item
DP	Document profile (Bloom filter)
UP	Interest profile of the user (Bloom filter)
$Snippet$	Summary of the document (title & synopsis with some/all query terms)
$Freq$	Frequency of a feedback item for Q as managed by node $P(Q)$ (moving average)
T_{first}	First arrival time of a feedback item for a given query Q on $P(Q)$
T_{last}	Last arrival time of a feedback item for a given query Q on $P(Q)$

Table 6.1 – Notations.

6.3.2 Collecting interest feedback

When a user browses the result list for a query, the title, document reference, and snippet help her select the most relevant documents w.r.t. her query and her interests. The action of accessing some document following a query produces a feedback information item. It represents an implicit *vote* for a document that the user, given her implicit expectations (as summarized by her user profile UP), deemed interesting for the query. The following information is tracked and forms an $RFitem$: (1) the original query Q, (2) the document reference D, e.g., a URL, (3) the local interest profile UP of the user after it has been updated with keywords from Q and the snippet, and (4) the snippet of the document, when available. Elements that are not accessed are simply ignored.

6.3.3 Managing repositories

The repository for a query Q is maintained by a specific node $P(Q)$ in the system. Section 6.4 explains how this node is reached and how the load for popular queries is dynamically shared amongst several nodes. Managing a repository for some query Q consists of two operations: (1) the management of the relevance feedback information received for Q, and (2) the generation of the results to be sent to a user submitting a request for Q.

We maintain one entry per tuple (Q, D) in the storage. The entries contain additional information $(DP, Snippet, Freq, T_{\text{first}}, T_{\text{last}})$, which are used for various tasks: sorting query results, storage management and garbage collection. Upon arrival of a new RFitem $(Q, D, UP, Snippet)$ at time t (see arrows labeled 2 in Figure 6.2), if an item (Q, D) already exists, it is updated by computing the union of the DP and UP bloom filters, updating the frequency, and setting T_{last} to t; otherwise, a new item is created and initialized using the content of the new RFitem.

6.3.4 Item ranking

When the node $P(Q)$ receives a request under the format (Q, UP) (see arrows labeled 1 in Figure 6.2), the storage manager extracts RFitems from the list associated with query Q and sorts them according to the similarity score w.r.t. the user profile (i.e., $Sim(UP, DP)$) and to the frequency. The resulting ranked list of document descriptors $(URL, Snippet)$ is then sent back to the user.

To ensure that a user profile UP provides sufficiently meaningful information to rank

search results according to the user's interests, we use on the client side a threshold that specifies the minimum number of distinct documents from the ongoing search session that must be embedded in the user profile for it to be sent along with the query. This helps ensure a minimum level of coherence in the results returned by CoFeed and avoids spending bandwidth and resources when no gain can be expected from the ranking information.

6.3.5 Garbage collection

Clients are continuously inserting new feedback information in the system. The storage on each node may be limited. A garbage collection mechanism allow to reclaim periodically some storage space while making sure that the most important information is preserved. Whenever a predefined limit for storage size has been reached (or when no further resources are available), items are pruned out based on the following rules, with decreasing order of priority: (1) frequency of item updates and last update time; (2) popularity thresholds; (3) utility of items for constructing results list.

6.4 Distributed Storage System

This section presents the design rationale of CoFeed's distributed storage system for managing repositories and allowing efficient processing of ranking and feedback insertion requests. We describe the resulting architecture and focus specifically on its two key features, routing and load balancing mechanisms.

As previously mentioned, our objective in the design of CoFeed is to support large populations of clients, each submitting many requests. To avoid the prohibitive cost of *scalable* centralized solutions (e.g., high traffic server farms), we propose a decentralized approach in which a set of nodes cooperates to provide the service. These nodes may be provided by ISPs or participating institutions (e.g., universities) that collectively share the processing load. The growth of the numbers of these nodes will follow the number of clients and allows solving the bootstrap problem from a resource provisioning perspective. The repository associated with a query is under the responsibility of a specific node in the system, but high loads are shared amongst several nodes. This node is located by using an efficient key-based routing protocol, which is described below.

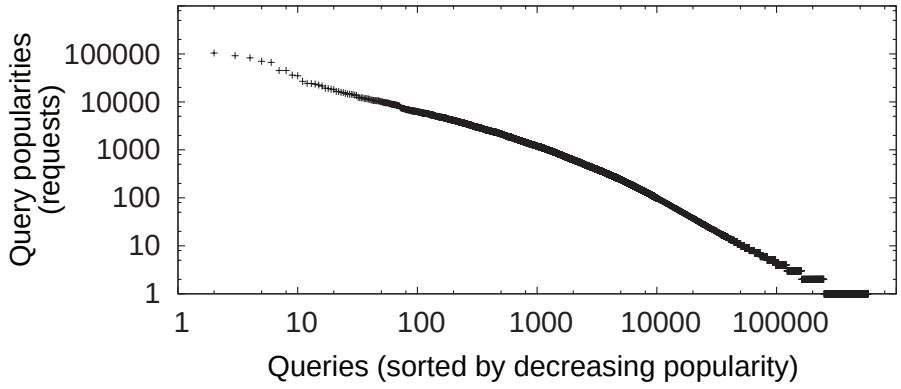


Figure 6.3 – Popularity distribution of queries in a dataset from AOL [102] follows a typical Zip-like [16] distribution.

A challenging aspect when designing the CoFeed distributed infrastructure is that the popularity distribution of queries is typically very sparse (that is, distributed according to a power law). This means that a small subset of the queries is requested extremely often while the vast majority is only rarely requested. This can be observed in Figure 6.3, where the popularity of requests from a representative query dataset from AOL [102] is plotted in decreasing order (note the logarithmic scales). Given the high skew in the distribution, one must ensure that popular queries do not overload specific nodes in the infrastructure. To protect against such scenarios, we have designed adaptive load balancing mechanisms to dynamically offload nodes experiencing too much incoming load. These mechanisms rely neither on fixed load threshold parameters nor manual tuning (see Section 6.4.2).

6.4.1 Routing

Each query Q is associated with a node $P(Q)$. This node stores the repository associated to Q: document references, relevance tracking and interest profiling information. Our overall design is a specialized form of a distributed hash table (DHT). It associates a key-based routing layer (KBR) and a storage layer. The role of the KBR layer is to locate the node responsible for some query based on its key. To that end, it relies on a structured overlay (e.g., an augmented ring), where each node is assigned a unique identifier and the responsibility of a range of data items identifiers. In our case, each query Q has an identifier determined by hashing its terms to a key $h(Q)$. The node $P(Q)$ whose range covers $h(Q)$ is responsible for maintaining Q's repository and for providing the appropriate sorted set of document references when asked to by some remote node. During the routing process, on each routing step towards the destina-

tion, the storage layer can be notified by a `Transit` call that a message is transiting *via* the local node. It can in turn modify the content of this message, or even answer the request on behalf of $P(Q)$. This mechanism is used in our design to implement load balancing.

A typical DHT provides a *raw* put/get interface to the application. Elements are stored as *blocks* on the node responsible for their key, and also retrieved as blocks. Our design differs in the following important point: our storage layer does not store information *blindly*, but provides an interface and functionalities that are *specific to the storage and processing of ranking and feedback information*. This has a strong impact on the design of fault-tolerance and load balancing mechanisms.

We base our system on the routing layer of Pastry [111], known for its stability and its performance (small number of hops, usage of network distance for choosing neighbors, etc.). In Pastry, nodes are organized in an augmented ring and maintain routing tables of size $O(log_b N)$, where b is a system parameter (keys are expressed in base b). Greedy routing succeeds in at most $O(log_b N)$ steps. When routing a request to its destination, each intermediary node selects as the next hop a node from its routing table with an identifier that has a longer common prefix with the target key than itself. As each routing step "resolves" at least one digit, at most $d = O(\log_b N)$ routing steps are required. An interesting property of such a greedy routing strategy is that routing paths towards a destination converge to the same set of nodes, and do so with an increasing probability as they get closer to the destination: the more digits have been resolved, the less nodes remain that have a longer common prefix with the target key. Routes from all nodes to some key in the network collide in the last hops. The *path convergence* property is particularly useful for the design of load balancing mechanisms [104, 119], as described next.

6.4.2 Load balancing

CoFeed needs to manage large numbers of users simultaneously and support the storage and access to repositories in a scalable manner. The sparseness of query popularities is the main problem, as nodes responsible for storing most popular queries may receive unbearable amounts of traffic.

When some node $P(Q)$ gets overloaded by requests to a popular query Q, it replicates its responsibility for managing information and answering requests related to Q. A wide range of techniques has been proposed for balancing load in structured overlays (e.g., [90, 104, 112, 119]). All these proposals however target scenarios where the number of accesses is much greater than the number of updates to the data. These

systems support access to non-mutable data by placing replicas on nodes that lie on the path towards its key.

Our system requirements are different. First, the amount of writes (insertion of interest tracking information) and the amount of reads (queries) are of the same order. Caching only read accesses is thus not possible: routing every insertion for a query Q to the node $P(Q)$ would involve notifying all copies, resulting in a load similar to the one avoided by caching access requests. It is thus necessary to also cache insertions, that is, to allow copies of information about a query to be modified *independently* from the "master" copy. We call such a copy a *delegate*: a replica onto which modifications are possible with only loose synchronization to its master copy. Second, queries are very dynamic by nature (e.g., a little-known personality can suddenly become famous and trigger millions of searches). Therefore, load balancing needs to be *reactive*, i.e., be able to initiate and cancel delegation dynamically as a function of the actual load.

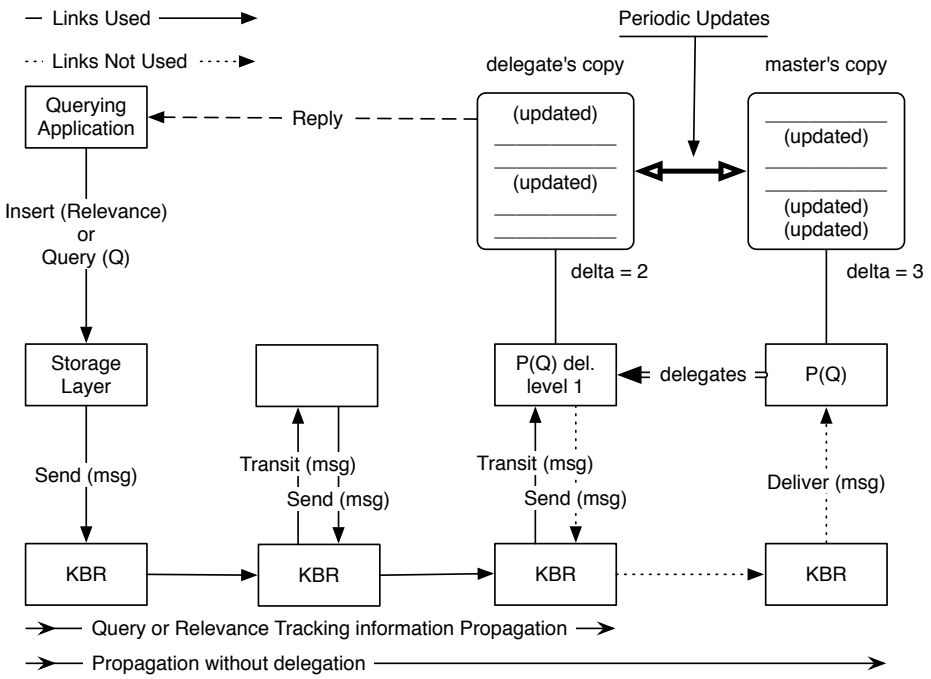


Figure 6.4 – Principle of the delegation mechanism.

Figure 6.4 presents the principle of delegation: a request (either for ranking or for insertion) is sent by the node on the left side and is routed towards the node $P(Q)$ on the right side. As the next to last node on the path is a delegate of $P(Q)$ for Q, it notices that a request for Q is going through its KBR layer and intercepts it. It replies on behalf of $P(Q)$ or inserts the information in its local copy. Periodic synchronization takes place between the delegates and their delegator (which may itself be a delegate).

Algorithm 2: Node p's periodic (Δ_{del}) auditing of incoming links.

SET: $cand \leftarrow \{c \in p.\text{in} \mid p.\text{in}_c \geq \sum_{x \in p.\text{in}} p.\text{in}_x / |p.\text{in}|\}$
foreach $c \in cand$ *(in parallel)* **do**
  retrieve $p.\text{load}_x$ from c
if $p.\text{load}_p > \gamma_{\text{del}} \times \sum_{c \in cand} p.\text{load}_c / |cand|$ **then**
  `// Details of logging ommited for brevity`
  during time Δ_{log}, log requests from nodes $cand$
  foreach $c \in cand$ **do**
    $c.q, c.q_{\text{load}} \leftarrow$ most frequent query from c, and its associated load
  $p.\text{load}_{\text{avg}} \leftarrow \left(p.\text{load}_p + \sum_{c \in cand} p.\text{load}_c\right) / \left(|cand| + 1\right)$
  choose $d \in cand$ that yields the minimal $|p.\text{load}_d(d.q \rightarrow d) - p.\text{load}_{\text{avg}}|$
  if $d.q_{load} > \sum_{x \in p.in} p.\text{in}_x \times \xi_{del}$ **then**
    send a copy of the repository for query $d.q$ to d
    delegate $d.q$ to q

Delegates are chosen according to the auditing Algorithm 2, which is run periodically by each node to evaluate its need for delegation. Table 6.2 gives the default parameter values used by the algorithm, as well as the notations used in the pseudocode.

The periodic auditing of the local load for deciding on a new delegation works as follows. $P(Q)$ keeps a counter $p.\text{in}_x$ of the number of requests received on each of its incoming links $p.\text{in}$, labeled by the previous hop x. Note that p does not maintain information about which query was targeted, as the role of this lightweight *passive monitoring* is only to detect load imbalance and not to spot their origin. All nodes in $p.\text{in}$ that sent more than the average load received on all p's incoming links are asked for their own incoming load, normalized to the period Δ_{del}. This information is stored in $p.\text{load}_x$ for node x.

The auditing of nodes for delegation is done only if sufficient imbalance is detected between the incoming load on node p and the load experienced by nodes in the *cand* (candidates) set. The imbalance threshold is γ_{del}: a value of 180% indicates that p has to handle more than 80% more requests than the average of *cand* nodes being investigated for possible delegation. If some imbalance is detected, the node enters a logging phase (*active monitoring*) in which the requests received from *cand* are recorded. This phase does not have to be as long as the passive monitoring phase, as only the most requested queries are of interest to p for deciding on a delegation, and those are likely to occur in great quantity even in a short period. Then, the most popular query received from each node $c \in cand$ is evaluated as a potential target for delegation. Basically, we select as delegate a node such that, when ignoring the most popular set of request for the same query coming from that node, the difference between the load experienced by

Constants		**Value**
Δ_{del}	Auditing period	15mn
Δ_{log}	Request logging period	5mn
γ_{del}	Imbalance tolerance before delegating	180%
ξ_{del}	Minimum relative gain for delegation decision	10%

Notations	
$p.\text{in}$	All "last-hop" nodes that sent some request to p during last period Δ_{del}
$p.\text{in}_x$	Number of requests p received from x during last period Δ_{del}
$p.\text{load}_x$	Inc. request load at x as known to p

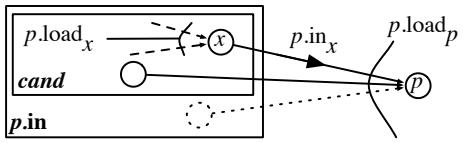


Table 6.2 – Delegation: constants and notations.

p and the average load experienced by the nodes in *cand* is minimal. Said differently, the goal is less to unload p than to evenly distribute the processing load on all nodes. Moreover, in order to prevent oscillations of delegations and un-delegations, p requires that at least ξ_{del} percent of its load will be handled by the new delegate.

When the delegation of a query by node d is decided, p sends a copy of the repository it has for the delegated query and instructs d to handle requests on its behalf. The cost of sending a delegation depends only on the size of the repository, which is typically small (in the order of a few kilobytes).

Delegates can in turn use this mechanism for re-delegating Q: the master copy on $P(Q)$ and its delegates form a tree. Synchronization between the copies is performed periodically when the number of changes, denoted *delta* in Figure 6.4, reaches a configurable threshold. Pair-wise synchronization is used to aggregate the two copies in a new list,

either by inserting "new" elements in the master list or by re-ranking the union of the two lists and keeping the k highest items. This list is then forwarded along the tree, resetting all *delta*s to 0.

Delegations are revoked by similar mechanisms: a node can revoke a delegation, based on the observation of requests load, either if it receives notably more requests than the other node for which it is a delegate, or if the revocation of the delegation helps balancing the load between a delegate and its delegator (i.e., the mean incoming load for both nodes gets closer to the average load observed by the delegate). This process uses hysteresis-based threshold values to avoid oscillations: the threshold for triggering delegation is higher than the threshold used for revoking one.

6.5 Evaluation

In this section, we evaluate CoFeed using two methods. Both use an actual implementation of the system. First, we assess the validity of interest profiling by running it against user behavior models. Second, we evaluate the performance and effectiveness of the infrastructure itself by observing the peak performance on a single node and the scalability in terms of managed elements, as well as distributed aspects: performance of routing, load balancing and reactiveness to dynamically changing loads.

Experiments were conducted on a cluster of 11 dual-core computers, each with 2 GB of main memory and running GNU/Linux. In experiments that do involve large number of nodes but no time-based performance measurements, each machine of the cluster executes *multiple processes* that represent different nodes. Naturally, for experiments that evaluate the performance of a single node *w.r.t.* time or peak performance, machines are used exclusively by one process. The implementation is based on a combination of C and Lua deployed using the SPLAY infrastructure.

6.5.1 User-centric ranking effectiveness

We first evaluate the effectiveness of interest-based profiling and ranking to actually report better tailored results to the user, especially in the case where this user issues a request for ambiguous query terms. To that extent, we developed both a synthetic data distribution model and a user behavior model. We do not consider distributed system aspects in this first part of the evaluation and assume that one node replies to all requests coming for one particular query (i.e., there is no use of load balancing). Our evaluation metrics are the ranks of elements of interest for the user, given her interest domain, with and without interest profiling.

We consider a set of U users $u_1, u_2, \ldots$, interested in a set of queries $Q = q_1, q_2, \ldots$ (e.g., "java", "jaguar", etc.). All these terms are ambiguous, and are associated to a set of documents (or elements) belonging to two or more *interest domains* chosen amongst $D = d_1, d_1, \ldots$. The actual number of domains $dom(q_i)$ for one query q_i is determined randomly using a power-law distribution: $dom(q_i) = 1 + extra(q_i)$, with $\Pr[extra(q_i)] \propto extra(q_i)^{-\alpha_{\text{dom/query}}}$. This means that most queries are associated with documents along 2 domains, a smaller set with documents over 3 domains, an even smaller with 4. No query is associated to more than 4 domains, and the parameter $\alpha_{\text{dom/query}}$ determines the skewness of this distribution. Each domain has a popularity, which is also determined using a power-law distribution: $Pr[d_i \in D] \propto i^{-\alpha_{\text{dompop}}}$. For each query q_i, the $dom(q_i)$ domains are selected according to this *domain popularity* distribution. Each user is interested in one single domain, also selected according to the same *domain popularity* distribution, and issues requests for elements related to this domain.

We consider a set of documents (or elements) $E = e_1, e_2 \ldots$, each of which is associated with one single interest domain chosen according to the domains' popularity distribution. For each domain d_i we create a list of documents $E(d_i)$, which is used as follows to generate a set of elements at each repository. Each query q_i is associated with a sorted set of 100 documents $E(q_i)$ representing the repository's content. Each element in this set is dedicated to one of the domains for which q_i is associated, chosen according the domain popularity distribution. The elements of the set are then filled by using a randomly picked and shuffled subset of $E(d_i)$. We use the values in Table 6.3 for the parameters of the workload.

Each document is associated with some text that represents the *content* of the document. This text is composed of a random number of keywords (between 15 and 30) chosen among queries from the domains associated with the document. To simulate the fact that the snippet returned by a centralized search engine for a given document will vary according to the search keywords (e.g., it highlights the sentences that surround the occurrence of the keywords in the original document), the snippet is generated as a random subset of 5 to 7 keywords forming the document content. One such snippet is generated initially for each query a document is attached to.

The search and access behavior of users is modeled using two phases. In a first phase, each user issues requests for queries that are attached to her interest domain and receives the list of elements as it is stored in the repository (i.e., without using interest-based ranking). This process continues until the user has sent at least 100 interest feedback items to the system (by simulated clicks on some of the returned results). This first phase helps construct the user and document profiles.

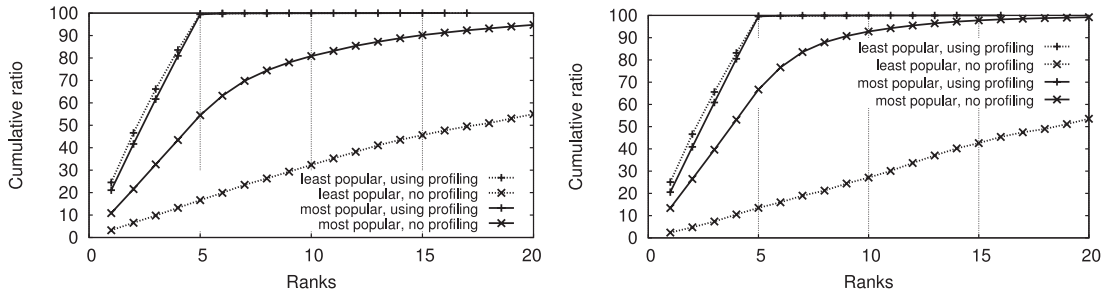
Name	Value	Role
$\|U\|$	500	Number of users
$\|Q\|$	2,000	Number of queries
$\|D\|$	20	Number of interest domains
$\|E\|/\|D\|$	400	Number of documents/elements per domain
$\alpha_{\text{dom/query}}$	1	Distribution of the number of extra domains per query
α_{dompop}	0.8	Distribution of the popularity of interest domains

Table 6.3 – Workload parameters.

We simulate the behavior of a user interested in the domain d receiving a list of items for some query q as follows. The user favors elements that are (1) higher up in the list, and (2) related to domain d. To model this behavior, we choose accessed elements according to a power-law distribution of the ranks in the list, with $Pr[accessing\ i^{th}\ element] \propto i^{-0.8}$, and we drop links that are not in d with probability 80%. In other words, there is a 20% chance that a user accesses some links that are not in her interest domain. This accounts for some "pollution" in the user and document profiles that is representative of real users' behaviors.

In a second phase, we compare the impact on the lists received by the users for their queries, of the use of the profiles and interest-based ranking. This allows us to evaluate whether the user profiling helps in leveraging links interesting to the user by ranking them higher.

For our evaluation, we consider two sets of domains: the 25% most popular ones (ranked 1 to 5) and the 25% least popular ones (ranked 16 to 20). We consider all the requests made by users that are interested in any of the domains of each set. For each such request, we examine the ranks of items that belong to the corresponding interest domain. We consider the ranks of the first 5 elements in the returned lists that are of the correct domain: the higher these 5 elements are in the list, the more effective the search mechanism is from the user point of view. We compare the distribution of these ranks both when using the direct result from the simulated search engine, and when using CoFeed.



(a) 25% least and most popular domains **(b)** 5% least and most popular domains

Figure 6.5 – Impact of interest-based profiling.

Obviously, there are more users interested in the 25% more popular interest domains than in the 25% least popular ones, and elements that are in the latter are ranked lower in the list returned by the centralized search engine model (being attached to an ambiguous query, they compete for positions in the list with at least one more popular domain). The goal of CoFeed is to promote links that are related to the user's domain of interest toward the first positions of her tailored list.

Figure 6.5a shows the cumulative distribution of the rank in the returned list for these first 5 elements, both when CoFeed interest-based ranking is used and when it is not, considering the 25% most/least popular elements. We observe that elements for the popular domains are already ranked higher than elements for the least popular domains: the median of the ranks of elements for popular domains is 5, while it is about 20 for unpopular domains. It follows that for both sets, CoFeed's ability to promote in the list the elements that are really of interest to the user is real, as in these representative sets, a vast majority of such elements appears in the first 5 ranks of the list. Figure 6.5b presents a similar plot, but when considering the 5% most/least popular set of the interest domains. We observe similar results, with unpopular domains ranked much higher in the list for the users who want them.

6.5.2 Repository peak performance

Next, we observe the performance of our prototype by running a single repository $P(Q)$ on a single machine (Core 2 Duo processor at 2.4 GHz with 2 GB memory) submitting synthetic request loads in a synchronous manner: we therefore achieve the highest possible throughput of requests that can be handled by one node in the system. We do not limit the size of the repository, as we want to highlight the relative cost of

inserting feedback information and ranking elements as a function of the number of stored elements.

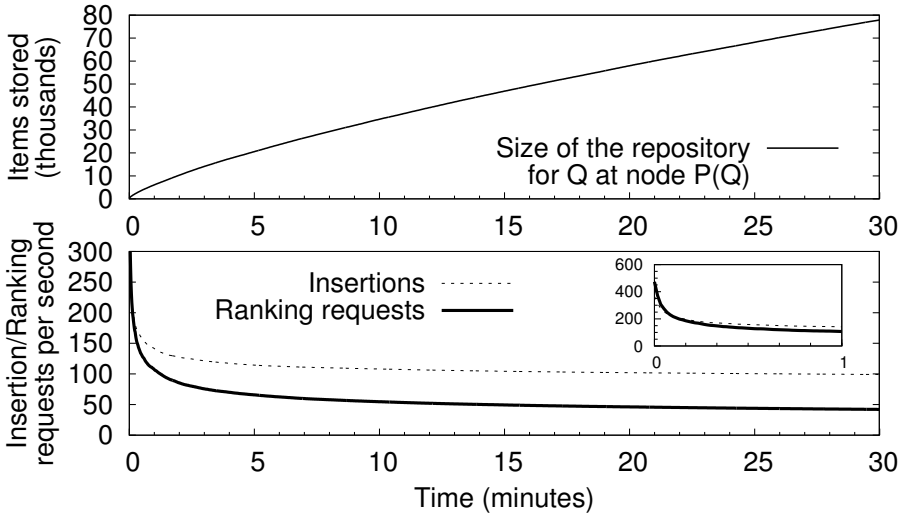


Figure 6.6 – Performance, single repository: max. possible load vs. repository size.

Figure 6.6 presents the maximal throughput evolution for insertions and ranking requests submitted alternatively. We observe that for reasonable repository sizes (up to 8,000 elements, which we expect to be the common case in practice), the throughput is consistently higher than 100 requests served per second. Note that the costs for one single request increase logarithmically in the size of the repository. The throughput still achieves as many as 50 ranking and 100 insertion requests per second with 30,000 items in the repository.

6.5.3 Routing layer

We measured the distribution of route lengths at the KBR layer for various system side. As expected [111], the distribution of route lengths is balanced around a low average route size (3.7 for 128 nodes, 5.7 for 4,096 nodes, 6.5 for 16,3984 nodes), which grows logarithmically in the system size.

6.5.4 Delegation-based load balancing: efficiency and reactiveness

Figure 6.7 shows a 3-days experiment using real request load from AOL [102].[1] The experiment evaluates two aspects: a bootstrap phase with no *dramatic* change in the user interest, showing the balancing process with stable popularity distributions, and a second phase with a previously unknown query Q_{pop} (artificially added to the AOL data set) suddenly generating a massive load in the system followed by a massive loss of popularity. During this time period, the associated $P(Q_{\text{pop}})$ has to efficiently tackle the massive and sudden load imbalance.

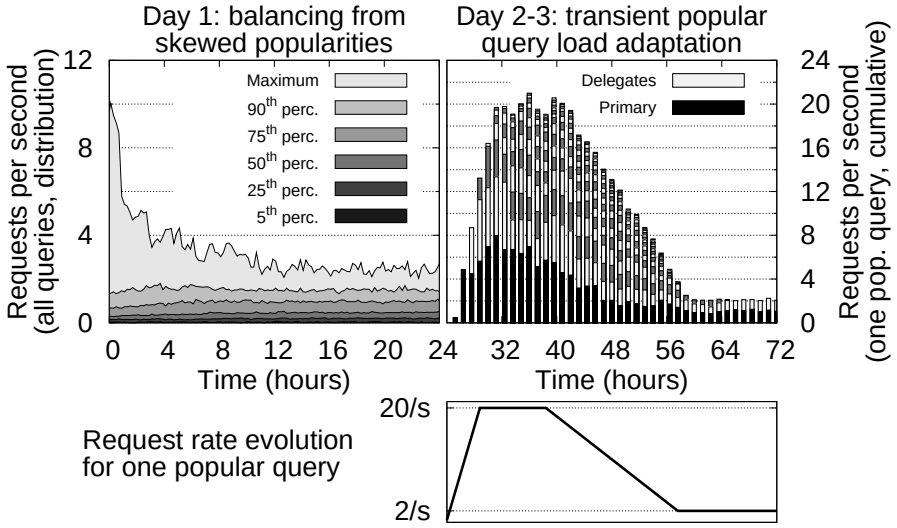


Figure 6.7 – Evaluation of the delegation mechanism reactiveness and efficiency.

The first day (on the left) presents the evolution of the distribution of the request load on all 1024 nodes, as delegation progressively takes place. The system starts up without bootstrap at time $t_0 = 0$ hours and initially no delegation is made. The evolution of the load distribution is presented by stacking up *percentiles*: the median load is thus represented by the 50$^{\text{th}}$ percentile and the maximal load by the lightest shade of gray. We observe that, from an initial high imbalance (where some nodes receive 10 times more load than 50% of all nodes), the system quickly converges to a reasonable imbalance (the most loaded node receives approximately twice as many requests than

[1] Unfortunately we could not use this data set for our evaluation of the profiling and ranking effectiveness because it lacks the necessary feedback information.

50% of the nodes). Further balancing could probably be achieved by modifying γ_{del} and ξ_{del} (see Section 6.4.2), but the small extra gain would likely not compensate for the additional synchronization messages necessary to more evenly balance the load.

The two last days (on the right) present the reactiveness of the system *for one single and suddenly popular query* Q_{pop} (while the first day presented results for *all* queries). At time t_s=24 hours, randomly chosen nodes in the system start issuing requests for Q_{pop}. The rate (shown in the bottom-right graph) reaches 20 requests per second in 4.8 hours, i.e., 10 times more than the median overall load at each node; it then remains constant for 9.6 more hours, before decreasing during 19.2 hours. The upper graph presents the load for Q_{pop} at $P(Q_{\text{pop}})$ (black bars) and its delegates (gray bars). Each bar represents the load and number of delegates at the end of a 70-minutes observation period. We observe that the number of delegates follows the popularity trend closely, in both directions (gain and loss). Furthermore, the load at $P(Q_{\text{pop}})$ experiences a small increase in the beginning but remains very low and stable when delegation is active. While some delegates may have only a very small portion of the load, they are still serving 1 or 2 queries per second, i.e., about the median load at all nodes. This is due to delegation decisions being made not based on fixed threshold but on the comparison of the loads of several nodes. Similarly, some delegates have a higher load than others but the imbalance remains within the limits imposed by the γ_{del} and ξ_{del} parameters.

6.6 Related Work

Many of the research efforts on P2P Web search focus on decreasing the bandwidth consumption as compared to a centralized approach [21, 83, 91, 122]. However, none of these P2P systems has yet succeeded in gaining sufficient popularity as they all suffer from the bootstrapping problem. CoFeed avoids this problem by leveraging existing search engines and providing added value to the user.

The personalization of search results for a user based on her interest profile was studied by [123, 124] but not exploited in the context where knowledge is collaboratively built and aggregated. The use of social annotations (e.g., from bookmarking platforms such as *del.icio.us*) to improve Web search has been recently explored [20, 115]. Another example is the *PeerSpective* system [99], which leverages implicit interest between communities of users based on the posting of links from one page to the other on social networks (e.g., FaceBook, MySpace, etc.). Such services operate in a centralized way and require intervention from the user to bookmark and annotate accessed items, which restricts them to a small subset of *power users*.

Our approach is more similar to the Chora [58] and Sixearch [17] systems, which also use decentralized architectures for sharing and leveraging user search experiences. CoFeed differs from these systems in several ways, notably they do not use interest profiling nor do they target information diversity.

A decentralized storage specifically designed for P2P Web search has been proposed in [73] for term frequency-inverse document frequency (TF-IDF). Unlike CoFeed, this system does not provide any mechanism for handling the skew in the popularity of queries, and it does not deal with the terms extraction nor use user-centric information to answer the queries.

Lopes *et al.* have proposed in [90] a storage architecture for large data on top of a DHT, using B+-trees to balance the storage load over several nodes. This architecture was designed for TF-IDF and only supports non-mutable data. Several other systems use the inverse routing paths convergence property, notably for load balancing [119] and or for replication and performance [104].

6.7 Perspectives

This work opens a set of interesting perspectives, out of which we would like to highlight some challenges associated with the management of user profiles over time. Currently, we use a simple threshold-based mechanisms for deciding when a profile has been bootstrapped with enough items and can be used for representing the interest of the client. We do not use a threshold on the maximal number of items that will be encoded in the profile itself: a new profile is simply created for each new browsing session and we assume that the interests of the user will be almost consistent throughout such a session, and a session be of limited time. This approach has some limitations. First, the necessary bootstrapping renders CoFeed useful for providing tailored search results only after some user activity, which may not be the case for each browsing session. Moreover, the information about the user's interest is not kept between search sessions. We will evaluate the possibility to re-use profiles amongst sessions. The decision of re-using or starting afresh a profile could be based on similarity metrics between sessions' profiles (selecting the profile, or the combination of profiles, that yield the best results according to the observed navigation and ranking requests). Finally, not having a threshold on the number of elements inserted into the profiles leads to the risk of loosing on the selectiveness of the profile and henceforth loosing on its ability to tailor ranking results. Our directions include using time-decaying profile construction, as well as the computation of selectiveness metrics when performing the ranking in order to assess that a profile is not representing a too-wide range of interests that prevent tailoring finely the results.

6.8 Summary

We have presented the architecture and building blocks ofthe novel CoFeed *collaborative ranking service* that can efficiently complement existing search engines. CoFeed leverages user-centric information such as interest profiling and relevance tracking in order to return search result lists tailored to the user interests. Collaborative ranking allows us to present tailored results to users, which can be more relevant especially when the user expectations do not follow the main trend. CoFeed combines methods for interest profiling and mechanisms to maintain *information diversity*. It builds on a support distributed P2P systems that combines classical key-based routing with an application specific storage layer. This layer proposes novel load balancing mechanisms based on the application needs and characteristics.

Chapter 7

Conclusion

> I would never die for my belief, because I might be wrong.
>
> Bertrand Russell

Building large-scale distributed systems is the next big challenge that a new generation of developers will have to face. Designing such systems is a highly complex task. Experience has shown that the gap between a pseudo code algorithm and a production-ready implementation is huge.

The current languages, simulators and testbeds are not sufficiently integrated. In order to use them efficiently, the effort is substantial and system specific: the work done and the lessons learned for a particular simulator or testbed has to be repeated to run a similar experiment on another one.

SPLAY has shown to be an innovative full stack distributed systems framework built from the ground up on highly modular, light and secure components. These technical choices permit us to develop distributed protocols implementations in record time. The abstraction it provides over existing testbeds minimises the effort of using them and avoids specific implementations. SPLAY have also proved that effective P2P research can be done without using simulation.

Many computing infrastructures could also be leveraged for activities related to distributed system design and evaluation (research, teaching, monitoring, etc.). Examples include networks of idle workstations in universities, research facilities and companies, or simply user-driven networks of personal computers. The highly configurable sandbox gives an additional advantage when running SPLAY on non dedicated computers.

This thesis made the following contributions.

First, we introduced a distributed infrastructure that greatly simplifies the prototyping, development, deployment, and execution of large-scale distributed applications and overlay networks. SPLAY also features several original properties, such as support for spontaneous overlays and embeddable algorithms.

Second, we showed how SPLAY applications can be concisely expressed with our specialized language. To illustrate and validate our approach, we have designed several well-known systems: Chord [121], Pastry [110], Scribe [34], SplitStream [35], BitTorrent [42], and Cyclon [129].

We also have implemented a number of classical algorithms, such Renyi-Erdös epidemic broadcast [47] and various types of distribution trees [25], whose expected behavior has been well studied and constitutes a good benchmark.

Third, we demonstrated experimentally that our system performs correctly and efficiently is real settings. Unlike Mace or P2, whose published experimental evaluation was performed on network emulators [127, 131], we deployed SPLAY on the whole PlanetLab [1]. Our system has been running for several months without encountering any major issue.

Fourth, we illustrated that SPLAY can be very useful to help crafting new distributed protocols by providing a very effective unified environment: we used the SPLAY framework as the primary tools to develop and validate the PULP protocol and also to build the backend of a collaborative ranking architecture. We are certain that in all these developments, SPLAY has greatly reduced the time spent and allowed us to focus more on algorithms.

Fifth, we are proud to see that SPLAY and SPLAYWEB have been successfully used since 2009 in the context of a graduate course on large scale distributed systems at the universities of Neuchâtel, Bern and Fribourg in Switzerland [108].

Finally, several papers have already used SPLAY with success: [52, 117, 92, 94, 128, 93, 96, 95].

7.1 Future Research

After I finished my work on SPLAY, several improvements have already been done in the framework.

SPLAY now features a distributed network topology attached to a deployment, similar to the one found in ModelNet. This topology permits to artificially add delays, bandwidth restriction and packet losses to each link. Contrary to ModelNet, this topology can be both usable in a clustered local environment or with nodes distributed on a real network and does not require dedicated traffic shaping nodes. Interestingly, the simplicity of SPLAY is retained as no modifications are required for the applications. This work has been published in [116].

A clock synchronisation mechanism (including for example the NTP library directly in splayd) would be valuable. The log system could generate entries with precise timing information directly from the node generating them (not affected by network latencies). Having 'sorted' logs is very helpful when trying to understand what was going on. Commands like `[START]` can also benefit of time synchronization.

The churn management has already been improved by being distributed between the controller and the splayds. Each splayd manages churn traces for its own nodes under the global monitor of the controller (if a splayd dies, the controller can still reallocate the job and the trace to another splayd) minimizing the impact of network delays. In the future, using the clock synchronization mechanism, the churn traces could be synchronized by a starting time. Having an higher precision can be important, especially when speeding up the traces to reduce the duration of experiments.

The churn language should be able to describe a distributions of the session time (for example using a Zipf's law) because all nodes have not a similar behavior facing it. Secondly, we should also distinguish between peers that disappear and are replaced by new ones and application that are just temporary shutdown and restarted later (a simple way to do it in collaboration with the application could be to not delete the storage between executions, another could be to suspend the process).

The deployment layer could also be extended to become more generic and support for instance deployment of architecture-dependant binary libraries.

Our RPC mechanism is built on top of TCP (stream oriented) or UDP (datagram oriented) layers. SPLAY would benefit of having a higher level abstraction layer using a MQ (Message Queuing) library like ØMQ [14]. This extra layer would minimize the amount of work related to resources and error management. A library like ØMQ would remove the complexity of managing transport layers while permitting to use most

of them, including IPC and multicast. The library would also provide connectivity patterns like fanout, pub/sub and task distribution.

Finally, one of the remaining drawback of SPLAY is the requirement to use Lua. Despite all its advantages, we know that learning a new language can be a barrier to its adoption. Using a more generic sandbox like Google NaCl seems a good way to address this limitation by supporting extra languages (C and C++ are directly supported and other languages like python and C# are already available).

SPLAY is publicly available from http://www.splay-project.org.

Publications

[49] Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, Etienne Rivière, Valerio Schiavoni, and José Valerio. "CoFeed: privacy-preserving Web search recommendation based on collaborative aggregation of interest feedback". In: *Software: Practice and Experience* (2011), n/a–n/a. ISSN: 1097-024X. DOI: 10.1002/spe.1127. URL: http://dx.doi.org/10.1002/spe.1127.

[50] Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, and Etienne Rivière. "Collaborative Ranking and Profiling: Exploiting the Wisdom of Crowds in Tailored Web Search". In: *Proceedings of DAIS'10: 10th IFIP international conference on Distributed Applications and Interoperable Systems*. Amsterdam, The Netherlands, 2010.

[51] Pascal Felber, Anne-Marie Kermarrec, Lorenzo Leonini, Etienne Rivière, and Spyros Voulgaris. "Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams". In: *Peer-to-Peer Networking and Applications* 5.1 (2012), pp. 74–91.

[78] Lorenzo Leonini, Etienne Rivière, and Pascal Felber. "P2P experimentations with SPLAY: from idea to deployment results in 30 min." In: *Proceedings of the Eigth International Conference on Peer-to-Peer Computing (P2P'08), demo sessions*. IEEE. Aachen, Germany, 2008.

[79] Lorenzo Leonini, Etienne Rivière, and Pascal Felber. "SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze)". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. Boston, MA, USA, 2009, pp. 185–198.

Bibliography

[1] http://www.planet-lab.org.

[2] http://www.telegeography.com.

[3] http://openmosix.sourceforge.net.

[4] http://www.kerrighed.org.

[5] http://shootout.alioth.debian.org.

[6] https://wiki.theory.org/BitTorrentSpecification.

[7] http://www.cs.berkeley.edu/~pbg/availability/.

[8] http://pypi.python.org/pypi/RestrictedPython/.

[9] http://freepastry.rice.edu.

[10] http://pdos.csail.mit.edu/chord/.

[11] http://www.crossflux.org/crcp/.

[12] http://ftp.ircache.net/Traces/.

[13] http://googleblog.blogspot.com/2008/11/searchwiki-make-search-your-own.html.

[14] http://www.zeromq.org.

[15] M.B. Abbott and L. Peterson. "A language-based approach to protocol implementation". In: *IEEE/ACM Trans. Netw.* 1.1 (Feb. 1993), pp. 4–19. URL: citeseer.ist.psu.edu/article/abbott93languagebased.html.

[16] Lada A. Adamic and Bernardo A. Huberman. "Zipf's law and the Internet". In: *Glottometrics* 3 (2002), pp. 143–150.

[17] R. Akavipat, L.-S. Wu, F. Menczer, and A.G. Maguitman. "Emerging semantic communities in peer web search". In: *P2PIR'06*.

[18] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. "Remote Control: Distributed Application Configuration, Management, and Visualization with Plush". In: *LISA'07*. 2007.

[19] D.P. Anderson. "Automated Protocol Implementation with RTAG". In: *IEEE Trans. Soft. Eng.* 14.3 (1988), pp. 291–300. ISSN: 0098-5589. DOI: http://dx.doi.org/10.1109/32.4650.

[20] Shenghua Bao, Guirong Xue, Xiaoyuan Wu, Yong Yu, Ben Fei, and Zhong Su. "Optimizing web search using social annotations". In: *WWW'07*.

[21] Matthias Bender, Sebastian Michel, Gerhard Weikum, and Christian Zimmer. "The Minerva Project: Database Selection in the Context of P2P Search". In: *Datenbanksysteme in Business, Technologie und Web* 65 (2005), pp. 125–144.

[22] Tim Berners-Lee and Robert Cailliau. *WorldWideWeb: Proposal for a HyperText Project*. Tech. rep. CERN, 1990. URL: http://www.w3.org/Proposal.

[23] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. "Understanding availability". In: *IPTPS*. Berkely, CA, USA, Feb. 2003.

[24] Danny Bickson and Dahlia Malkhi. "The julia content distribution network". In: *Proc. of WORLDS*. San Francisco, CA, USA, 2005, pp. 7–7.

[25] E.W. Biersack, P. Rodriguez, and P. Felber. "Performance Analysis of Peer-to-Peer Networks for File Distribution". In: *QofIS'04*. Sept. 2004, pp. 1–10.

[26] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. "Bimodal Multicast". In: *ACM Trans. Computer Systems* 17.2 (1999), pp. 41–88.

[27] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Commun. ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692. URL: http://portal.acm.org/citation.cfm?id=362692.

[28] Thomas Bonald, Laurent Massoulié, Fabien Mathieu, Diego Perino, and Andrew Twigg. "Epidemic live streaming: optimal performance trade-offs". In: *SIGMETRICS*. Annapolis, MA, USA, June 2008, pp. 325–336.

[29] Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation. 2007.

[30] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *PODC*. 2000, p. 7.

[31] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine". In: 30.1–7 (Apr. 1998), pp. 107–117. ISSN: 0169-7552. URL: http://www.elsevier.com/cas/tree/store/comnet/sub/1998/30/1-7/1921.pdf.

[32] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine". In: *Computer Networks and ISDN Systems* 30.1–7 (1998), pp. 107–117.

[33] Michael Burrows. "The Chubby Lock Service for Loosely-Coupled Distributed Systems". In: *OSDI*. 2006, pp. 335–350.

[34] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. "SCRIBE: A large-scale and decentralized publish-subscribe infrastructure". In: *IEEE J. Sel. Areas Commun.* 20.8 (Oct. 2002). Special issue on Network Support for Multicast Communications.

[35] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. "SplitStream: High-bandwidth multicast in a cooperative environment". In: *SOSP'03*. Oct. 2003.

[36] Mary-Luc Champel, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. "FoG: Fighting the Achilles' Heel of Gossip Protocols with Fountain Codes". In: *SSS*. Lyon, France, 2009, pp. 180–194.

[37] Ramesh Chandra, Ryan M. Lefever, Michel Cukier, and William H. Sanders. "Loki: A State-Driven Fault Injector for Distributed Systems". In: *DSN'00*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 237–242. ISBN: 0-7695-0707-7.

[38] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *PODC*. 2007, pp. 398–407.

[39] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: a distributed storage system for structured data". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI '06. Seattle, WA: USENIX Association, 2006, pp. 15–15. URL: http://dl.acm.org/citation.cfm?id=1267308.1267323.

[40] Kai Cheng, Limin Xiang, Mizuho Iwaihara, Haiyan Xu, and Mukesh M. Mohania. "Time-Decaying Bloom Filters for Data Streams with Skewed Distributions". In: *RIDE-SDMA'05*.

[41] Renato Lo Cigno, Alessandro Russo, and Damiano Carra. "On some fundamental properties of P2P push/pull protocols". In: *Proceedings of the 2nd International Conference on Communications and Electronic (HUT-ICCE)*. HoiAn, Vietnam, 2008, pp. 67–73.

[42] B. Cohen. *Incentives to Build Robustness in BitTorrent*. Tech. rep. http://www.bittorrent.org/, May 2003.

[43] Edited Jim Coplien, Douglas C. Schmidt, and Douglas C. Schmidt. *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. 1995.

[44] Jeffrey Dean, Sanjay Ghemawat, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI*. 2004, pp. 137–150.

[45] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic algorithms for replicated database maintenance". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*. Vancouver, Canada, Aug. 1987, pp. 1–12.

[46] Alexander Dupuy, Jed Schwartz, Yechiam Yemini, and David Bacon. "NEST: a network simulation and prototyping testbed". In: *Commun. ACM* 33.10 (1990), pp. 63–74. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/84537.84549.

[47] P. Erdös and A. Rényi. "On the evolution of random graphs". In: *Mat. Kuttató. Int. Közl.* 5 (1960), pp. 17–60.

[48] Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. "Lightweight Probabilistic Broadcast". In: *ACM Transaction on Computer Systems* 21.4 (Nov. 2003).

[49] Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, Etienne Rivière, Valerio Schiavoni, and José Valerio. "CoFeed: privacy-preserving Web search recommendation based on collaborative aggregation of interest feedback". In: *Software: Practice and Experience* (2011), n/a–n/a. ISSN: 1097-024X. DOI: 10.1002/spe.1127. URL: http://dx.doi.org/10.1002/spe.1127.

[50] Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, and Etienne Rivière. "Collaborative Ranking and Profiling: Exploiting the Wisdom of Crowds in Tailored Web Search". In: *Proceedings of DAIS'10: 10th IFIP international conference on Distributed Applications and Interoperable Systems*. Amsterdam, The Netherlands, 2010.

[51] Pascal Felber, Anne-Marie Kermarrec, Lorenzo Leonini, Etienne Rivière, and Spyros Voulgaris. "Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams". In: *Peer-to-Peer Networking and Applications* 5.1 (2012), pp. 74–91.

[52] Pascal Felber, Martin Rajman, Etienne Riviere, Valerio Schiavoni, and José Valerio. "SPADS: Publisher Anonymization for DHT Storage". In: *Peer-to-Peer Computing*. 2010, pp. 1–10.

[53] Luiz Henrique de Figueiredo, Roberto Ierusalimschy, and Waldemar Celes Filho. "The Design and Implementation of a Language for Extending Applications". In: *IN XXI SEMISH*. 1994, pp. 273–284.

[54] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Martin Mogensen, Maxime Monod, and Vivien Quéma. *Gossiping Capabilities*. Tech. rep. LPD-REPORT-2008-010. EPFL, 2008.

[55] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. "Peer-to-Peer membership management for gossip-based protocols". In: *IEEE Transactions on Computers* 52.2 (Feb. 2003), pp. 139–149.

[56] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *SOSP*. 2003, pp. 29–43.

[57] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002), pp. 51–59. URL: http://dx.doi.org/10.1145/564585.564601.

[58] Halldor Gylfason, Omar Khan, and Grant Schoenebeck. "Chora: Expert-based P2P Web Search". In: *AAMAS'06*.

[59] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *In USENIX Annual Technical Conference*.

[60] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. "The Implementation of Lua 5.0". In: *J. of Univ. Comp. Sc.* 11.7 (2005), pp. 1159–1176.

[61] Sitaram Iyer, Antony Rowstron, and Peter Druschel. "Squirrel: a decentralized peer-to-peer web cache". In: *PODC'02*. New York, NY, USA, 2002, pp. 213–222.

[62] Elliot Jaffe, Danny Bickson, and Scott Kirkpatrick. "Everlab: a production platform for research in network experimentation and computation". In: *LISA'07*. Dallas, 2007, pp. 1–11. ISBN: 978-1-59327-152-7.

[63] Mark Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. "Gossip-Based Peer Sampling." In: *ACM Transactions on Computer Systems* 25.3 (2007).

[64] Flavio Paiva Junqueira and Benjamin C. Reed. "Brief Announcement Zab: A Practical Totally Ordered Broadcast Protocol". In: *DISC*. 2009, pp. 362–363.

[65] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *DSN*. 2011, pp. 245–256.

[66] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. "Randomized Rumour Spreading". In: *IEEE Proc. 41st Ann. Symp. Foundations of Computer Science (FOCS)*. Nov. 2000, p. 565.

[67] Srinivas Kashyap, Supratim Deb, K. V. M. Naidu, Rajeev Rastogi, and Anand Srinivasan. "Efficient gossip-based aggregate computation". In: *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM Press, June 2006, pp. 308–317.

[68] Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. "Probabilistic Reliable Dissemination in Large-Scale Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 14.3 (Mar. 2003), pp. 139–149.

[69] Anne-Marie Kermarrec and Maarten van Steen. "Gossiping in Distributed Systems". In: *ACM Operating System Review* 41.5 (2007).

[70] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. "NAT-resilient Gossip Peer Sampling". In: *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 360–367.

[71] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. "NAT-resilient Gossip Peer Sampling". In: *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 360–367. ISBN: 978-0-7695-3659-0. DOI: http://dx.doi.org/10.1109/ICDCS.2009.44.

[72] C. Killian, J.W. Anderson, R. Braud, R. Jhala, and A. Vahdat. "Mace: Language Support for Building Distributed Systems". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Diego, CA, June 2007.

[73] Fabius Klemm and Karl Aberer. "Aggregation of a Term Vocabulary for Peer-to-Peer Information Retrieval: a DHT Stress Test". In: *DBISP2P'05*.

[74] E. Kohler, M.F. Kaashoek, and D.R. Montgomery. "A readable TCP in the Prolac protocol language". In: *SIGCOMM Comput. Commun. Rev.* 29.4 (1999), pp. 3–13. ISSN: 0146-4833. DOI: http://doi.acm.org/10.1145/316194.316200.

[75] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Kenneth P. Birman, and Alan J. Demers. "Active and passive techniques for group size estimation in large-scale and dynamic distributed systems". In: *Journal of Systems and Software* 80.10 (2007), pp. 1639–1658. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2007.01.014.

[76] Leslie Lamport. "Paxos Made Simple, Fast, and Byzantine". In: *OPODIS*. 2002, pp. 7–9.

[77] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169.

[78] Lorenzo Leonini, Etienne Rivière, and Pascal Felber. "P2P experimentations with SPLAY: from idea to deployment results in 30 min." In: *Proceedings of the Eigth International Conference on Peer-to-Peer Computing (P2P'08), demo sessions*. IEEE. Aachen, Germany, 2008.

[79] Lorenzo Leonini, Etienne Rivière, and Pascal Felber. "SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze)". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. Boston, MA, USA, 2009, pp. 185–198.

[80] Bo Li, Yang Qu, G.Y. Keung, Susu Xie, Chuang Lin, Jiangchuan Liu, and Xinyan Zhang. "Inside the New Coolstreaming: Principles, Measurements and Performance Implications". In: *Proceedings of the 27th Conference on Computer Communications (IEEE INFOCOM)*. Apr. 2008, pp. 1031–1039.

[81] Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Lorenzo Alvisi, and Michael Dahlin. "BAR Gossip". In: *Proc. of 7th Symposium on Operating System Design and Implementation (OSDI '06)*. Nov. 2006, pp. 191–2004.

[82] Harry C. Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin. "FlightPath: Obedience vs. Choice in Cooperative Services". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. Dec. 2008.

[83] Jinyang Li, Boon Loo, Joseph Hellerstein, Frans Kaashoek, David Karger, and Robert Morris. "The Feasibility of Peer-to-Peer Web Indexing and Search". In: *IPTPS'03*.

[84] Meng Lin and Keith Marzullo. *Directional gossip: Gossip in a wide area network.* Tech. rep. CS1999-0622. University of California, San Diego, 1999.

[85] Meng Lin, Keith Marzullo, and Stefano Masini. "Gossip versus deterministic flooding: low-message overhead and high-reliability for broadcasting on small networks." In: *Intl. Symposium on Distributed Computing (DISC 2000)*. Toledo, Spain, 2000, pp. 85–89.

[86] Shiding Lin, Aimin Pan, Rui Guo, and Zheng Zhang. "Simulating Large-Scale P2P Systems with the WiDS Toolkit". In: *MASCOTS'05*. Washington, DC, USA: IEEE Computer Society, 2005. ISBN: 0-7695-2458-3. DOI: http://dx.doi.org/10.1109/MASCOT.2005.64.

[87] Xiaotao Liu, Jiang Lan, Prashant Shenoy, and Krithi Ramaritham. "Consistency maintenance in dynamic peer-to-peer overlay networks". In: *Computer Networks* 50.6 (2006), pp. 859–876.

[88] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. "Push-to-Pull Peer-to-Peer Live Streaming". In: *Proceedings of DISC 2007: 21st International Symposium on Distributed Computing*. Lemosos, Cyprus, 2007.

[89] B. Thau Loo, T. Condie, J.M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. "Implementing declarative overlays". In: *SOSP'05*. 2005, pp. 75–90.

[90] Nuno Lopes and Carlos Baquero. "Taming Hot-Spots in DHT Inverted Indexes". In: *LSDS-IR'07*.

[91] Toan Luu, Fabius Klemm, Ivana Podnar, Martin Rajman, and Karl Aberer. "ALVIS Peers: A Scalable Full-text Peer-to-Peer Retrieval Engine". In: *Proc of P2PIR'06*.

[92] Francisco Maia, Miguel Matos, Etienne Riviere, and Rui Oliveira. "Slead: Low-Memory, Steady Distributed Systems Slicing". In: *DAIS*. 2012, pp. 1–15.

[93] Francisco Maia, Miguel Matos, Rui Oliveira, and Etienne Riviere. "Slicing as a Distributed Systems Primitive". In: *LADC*. 2013, pp. 124–133.

[94] Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, and Etienne Riviere. "BRISA: Combining Efficiency and Reliability in Epidemic Data Dissemination". In: *IPDPS*. 2012, pp. 983–994.

[95] Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, and Etienne Rivière. "Lightweight, efficient, robust epidemic dissemination". In: *J. Parallel Distrib. Comput.* 73.7 (2013), pp. 987–999.

[96] Miguel Matos, Pascal Felber, Rui Oliveira, José Orlando Pereira, and Etienne Riviere. "Scaling Up Publish/Subscribe Overlays Using Interest Correlation for Link Sharing". In: *IEEE Trans. Parallel Distrib. Syst.* 24.12 (2013), pp. 2462–2471.

[97] Steve McConnell. *Software Estimation: Demystifying the Black Art*. Redmond, WA, USA: Microsoft Press, 2006. ISBN: 0735605351, 9780735605350.

[98] James W. Mickens and Brian D. Noble. "Exploiting availability prediction in distributed systems". In: *NSDI'06*. San Jose, CA: USENIX Association, 2006, pp. 73–86.

[99] Alan Mislove, Krishna P. Gummadi, and Peter Druschel. "Exploiting Social Networks for Internet Search". In: *HotNets'06*.

[100] Lucas Nussbaum and Olivier Richard. "Lightweight emulation to study peer-to-peer systems". In: *Concurr. Comput. : Pract. Exper.* 20.6 (2008), pp. 735–749. ISSN: 1532-0626. DOI: http://dx.doi.org/10.1002/cpe.v20:6.

[101] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. "Chainsaw: Eliminating Trees from Overlay Multicast". In: *IPTPS'05: the fourth International Workshop on Peer-to-Peer Systems*. Feb. 2005, pp. 127–140.

[102] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. "A picture of search". In: *InfoScale '06*. New York, NY, USA.

[103] Fabio Picconi and Laurent Massoulié. "Is there a future for mesh-based live video streaming?" In: *Proceedings of the Eigth International Conference on Peer-to-Peer Computing (P2P'08)*. Aachen, Germany, Sept. 2008.

[104] Venugopalan Ramasubramanian and Emin Gün Sirer. "Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays". In: *NSDI'04*.

[105] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. "A scalable content-addressable network". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 161–172. ISSN: 0146-4833. DOI: 10.1145/964723.383072. URL: http://doi.acm.org/10.1145/964723.383072.

[106] Robbert van Renesse, Yaron Minsky, and Mark Hayden. "A Gossip-Style Failure Detection Service". In: *Proc. of Middleware, the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Ed. by IFIP. The Lake District, United Kingdom, 1998, pp. 55–70.

[107] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera, and Jay Lepreau. "The Flexlab Approach to Realistic Evaluation of Networked Systems". In: *NSDI'07*.

[108] Etienne Rivière. "Simplifying Hands-On Teaching of Distributed Algorithms with SPLAY". In: *IPDPS Workshops*. 2012, pp. 1311–1316.

[109] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. "NATCracker: NAT Combinations Matter". In: *Proc. of ICCN'09: 18th International Conference on Computer Communications and Networks*. Vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 1–7.

[110] A. Rowstron and P. Druschel. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". In: *Middleware'01*. Nov. 2001.

[111] Antony Rowstron and Peter Druschel. "Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems". In: *Proc. of Middleware'01*.

[112] Antony Rowstron and Peter Druschel. "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility". In: *SOSP'01*.

[113] Alessandro Russo and Renato Lo Cigno. "Delay-Aware Push/Pull Protocols for Live Video Streaming in P2P Systems". In: *Proceedings of the IEEE International Conference on Communications (ICC)*. May 2010.

[114] Sujay Sanghavi, Bruce Hajek, and Laurent Massoulié. "Gossiping with Multiple Messages". In: *Proceedings of the 29th Conference on Computer Communications (IEEE INFOCOM)*. May 2007, pp. 2135–2143.

[115] Ralf Schenkel, Tom Crecelius, Mouna Kacimi, Sebastian Michel, Thomas Neumann, Josiane Xavier Parreira, and Gerhard Weikum. "Efficient Top-k Querying over Social-Tagging Networks". In: *SIGIR'08*.

[116] Valerio Schiavoni, Etienne Rivière, and Pascal Felber. "SplayNet: Distributed User-Space Topology Emulation". In: *Middleware*. 2013, pp. 62–81.

[117] Valerio Schiavoni, Etienne Riviere, and Pascal Felber. "WHISPER: Middleware for Confidential Communication in Large-Scale Networks". In: *ICDCS*. 2011, pp. 456–466.

[118] Bianca Schroeder and Garth Gibson. "Large-scale Study of Failures in High-performance-computing Systems". In: *DSN'06*. Philadelphia, PA, USA, June 2006.

[119] Sabina Serbu, Silvia Bianchi, Peter Kropf, and Pascal Felber. "Dynamic Load Sharing in Peer-to-Peer Systems: When some Peers are more Equal than others." In: *IEEE Internet Computing, Special Issue on Resource Allocation* 11.4 (2007), pp. 53–61.

[120] Raghav Srinivasan, Chao Liang, and Krithi Ramamritham. "Maintaining Temporal Coherency of Virtual Data Warehouses". In: *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1998, p. 60.

[121] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Trans. Netw.* 11.1 (2003), pp. 17–32. ISSN: 1063-6692. DOI: http://dx.doi.org/10.1109/TNET.2002.808407.

[122] Torsten Suel, Chandan Mathur, Jo-Wen Wu, Jiangong Zhang, Alex Delis, Mehdi Kharrazi, Xiaohui Long, and Kulesh Shanmugasundaram. "ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval." In: *WebDB'03*.

[123] Bin Tan, Xuehua Shen, and ChengXiang Zhai. "Mining long-term search history to improve search accuracy". In: *SIGKDD'06*.

[124] Jaime Teevan, Susan T. Dumais, and Eric Horvitz. "Personalizing search via automated analysis of interests and activities". In: *SIGIR-IR'05*. ISBN: 1-59593-034-5. DOI: 10.1145/1076034.1076111. URL: http://portal.acm.org/citation.cfm?id=1076111.

[125] P. Urban, X. Defago, and A. Schiper. "Neko: A Single Environment to Simulate and Prototype Distributed Algorithms". In: *ICOIN'01*. 2001, pp. 503–511. URL: citeseer.ist.psu.edu/article/urban01neko.html.

[126] Bhuvan Urgaonkar, Anoop George Ninan, Mohammad Salimullah Raunak, Prashant Shenoy, and Krithi Ramamritham. "Maintaining Mutual Consistency for Cached Web Objects". In: *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, Apr. 2001, pp. 371–380.

[127] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. "Scalability and accuracy in a large-scale network emulator". In: *OSDI'02*. Boston, Massachusetts, 2002, pp. 271–284.

[128] José Valerio, Pierre Sutra, Etienne Rivière, and Pascal Felber. "Evaluating the Price of Consistency in Distributed File Storage Services". In: *DAIS*. 2013, pp. 141–154.

[129] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays". In: *Journal of Network and Systems Management* 13.2 (June 2005).

[130] Yanyan Wang, Antonio Carzaniga, and Alexander L. Wolf. "Four Enhancements to Automated Distributed System Experimentation Methods". In: *ICSE'08*. Leipzig, Germany, May 2008.

[131] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. "An Integrated Experimental Environment for Distributed Systems and Networks". In: *OSDI'02*. Boston, MA, Dec. 2002, pp. 255–270.

[132] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *IEEE Symposium on Security and Privacy*. 2009, pp. 79–93.

[133] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. "CoolStreaming/DONet: A Data-driven Overlay Network for Efficient Live Media Streaming". In: *Proceedings of the 24th Conference on Computer Communications (IEEE INFOCOM)*. Mar. 2005, pp. 2102–2111.

[134] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1 (Jan. 2004), pp. 41–53.