Improving skewed data dissemination in structured overlays

Maeva Antoine

► To cite this version:

Maeva Antoine. Improving skewed data dissemination in structured overlays. Other [cs.OH]. Université Nice Sophia Antipolis, 2015. English. NNT : 2015NICE4054 . tel-01245077

HAL Id: tel-01245077
https://theses.hal.science/tel-01245077v1

Submitted on 16 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ NICE SOPHIA ANTIPOLIS

École Doctorale STIC

Sciences et Technologies de l'Information et de la Communication

THÈSE

pour l'obtention du grade de

Docteur en Sciences

de l'Université Nice Sophia Antipolis

Mention Informatique

présentée et soutenue par

Maeva ANTOINE

Improving Skewed Data Dissemination in Structured Overlays

Thèse dirigée par Éric Madelaine

Soutenue le 23 septembre 2015

Jury

Examinateur	Jean-Louis Pazat	INSA Rennes
Rapporteurs	Esther Pacitti	Université de Montpellier 2
	Pierre Sens	Université de Paris 6
Directeur de thèse	Éric Madelaine	INRIA Sophia Antipolis
Encadrant scientifique	Fabrice Huet	Université Nice Sophia Antipolis
Invitée	Françoise Baude	Université Nice Sophia Antipolis

Résumé

De nombreux systèmes Pair-à-Pair structurés gérant des données sont confrontés au problème du déséquilibre de charge entre machines. Ce phénomène s'est amplifié avec l'émergence du Big Data où de larges volumes de données, aux valeurs souvent biaisées, sont produits par des sources hétérogènes pour être stockés et/ou analysés, souvent en temps réel. Les systèmes doivent donc être capables de s'adapter aux variations de volume/contenu/provenance des données entrantes. Dans cette thèse, nous nous intéressons aux données RDF, un format du Web Sémantique. En partant du constat établi avec notre propre système de stockage RDF, nous proposons une nouvelle approche pour améliorer la répartition des données, basée sur l'utilisation de plusieurs fonctions de hachage préservant l'ordre naturel des données dans le réseau. Cela permet à chaque pair de pouvoir indépendamment modifier la fonction de hachage qu'il applique sur les données afin de réduire l'intervalle de valeurs dont il est responsable. Ainsi, nous montrons comment différentes fonctions de hachage peuvent être appliquées par différents pairs sans nécessiter que chacun connaisse toutes les fonctions utilisées et tout en maintenant la topologie et le routage cohérents. Plus généralement, pour résoudre le problème du déséquilibre de charge, il existe presque autant de stratégies qu'il y a de systèmes différents. En outre, les solutions proposées sont souvent couplées à une API qui leur est propre, ce qui rend difficile le portage d'une solution d'un système à un autre. Dans cette thèse, nous montrons que de nombreux dispositifs d'équilibrage de charge sont constitués des mêmes éléments de base, et que seules la mise en œuvre et l'interconnexion de ces éléments varient. Partant de ce constat, nous décrivons les concepts derrière la construction d'une API générique pour appliquer une stratégie d'équilibrage de charge qui est indépendante du reste du code. Puis, nous montrons comment l'API est compatible avec certains systèmes existants et leur solution d'équilibrage de charge. Mise en place sur notre système, l'API a un impact minimal sur le code métier et permet de changer une partie d'une stratégie sans modifier d'autres composants. Nous montrons aussi comment la variation de certains paramètres permet d'obtenir des différences significatives de résultat.

Abstract

Many structured Peer-to-Peer systems for data management face the problem of load imbalance between machines. With the advent of Big Data, large datasets whose values are often highly skewed are produced by heterogeneous sources to be stored and/or analysed, often in real time. Thus, distributed systems must be able to adapt to the variations of size/content/source of the incoming data. In this thesis, we focus on RDF data, a format of the Semantic Web. Based on the observations made on our own distributed platform for RDF data storage, we propose a novel approach to improve data distribution, based on the use of several order-preserving hash functions. This allows an overloaded peer to independently modify its hash function in order to reduce the interval of values it is responsible for. We show how different hash functions can be used by different peers without requiring that each peer knows all the hash functions used in the system, while maintaining the same network topology and the overlay consistent. More generally, to address the load imbalance issue, there exist almost as many load balancing strategies as there are different systems. Besides, the proposed solutions are often coupled to their own API, making it difficult to port a scheme from a system to another. In this thesis, we show that many load balancing schemes are comprised of the same basic elements, and only the implementation and interconnection of these elements vary. Based on this observation, we describe the concepts behind the building of a common API to implement any load balancing strategy independently from the rest of the code. We then show how this API is compatible with famous existing systems and their load balancing scheme. Implemented on our distributed storage system, the API has a minimal impact on the business code and allows to change only a part of a strategy without modifying its other components. We also show how modifying some parameters can lead to significant improvements in terms of results.

Remerciements

Je tiens à remercier mes encadrants de l'équipe SCALE, Françoise Baude, Fabrice Huet et Eric Madelaine, pour leurs enseignements sur le monde de la recherche.
Un grand merci également à Laurent Pellegrino, avec qui ce fut un véritable plaisir de travailler et sur qui j'ai toujours pu compter depuis mon arrivée à l'INRIA.
Je n'oublie pas non plus mes collègues doctorantes Justine Rochas et Sophie Ge Song avec lesquelles j'ai eu l'occasion de collaborer.
Enfin, merci aux membres du jury, Esther Pacitti, Pierre Sens et Jean-Louis Pazat, d'avoir accepté d'accorder un peu de leur temps à l'évaluation de mes travaux.

Table of Contents

List of Figures **xi**

List of Listings **xiii**

List of Tables **xv**

1 Introduction **1**

1.1 Motivation and Problem Definition 1
1.2 Outline and Contributions 3

2 Background **7**

2.1 Semantic Web 8
2.1.1 RDF data model 9
2.1.2 SPARQL query language 11
2.2 Peer-to-Peer Systems 13
2.2.1 Overview 13
2.2.2 CAN 14
2.2.3 Chord 17
2.3 The EventCloud RDF Store 19
2.3.1 Architecture 19
2.3.2 Unicode 21
2.3.3 Subscription matching 22
2.3.4 Optimal broadcast algorithm 23
2.4 Other Existing RDF Storage Systems 24
2.4.1 Centralized approaches 26
2.4.2 Distributed approaches 26

2.5 Load Balancing Challenge 28

3 Existing Load Balancing Strategies 31

3.1 Load Balancing in non-P2P Systems 33

3.1.1 NoSQL 33

3.1.2 Stream processing 34

3.1.3 Cloud 35

3.2 Load Balancing in Structured P2P Systems 36

3.2.1 Load balancing for RDF data 37

3.2.2 Virtual servers reassignment 39

3.2.3 Data/Node replication 41

3.2.4 Keys reassignment 45

4 Variable Hash Functions 49

4.1 State of the Art 50

4.1.1 Multiple hash functions for popular items 50

4.1.2 Multiple hash functions for data dissemination 53

4.2 Principles and Interest of our Approach 55

4.2.1 CAN for data storage 57

4.2.2 Skewed data, skewed distribution 59

4.2.3 Variable hash functions for overloaded zones 60

4.3 Hash Function Update Process 64

4.3.1 Update propagation 64

4.3.2 Bound reduction in a torus-like topology 67

4.3.3 Optimal update propagation in a torus-like CAN 68

4.3.4 Concurrency and message filtering 71

4.3.5 Data movement and lookup 73

4.3.6 Main rules 77

4.4 Summary 79

5 Generic API for Load Balancing 81

5.1 Motivation 81

5.2 Load Balancing Differentiators 83

5.3 API Presentation 88

5.3.1 High-level abstractions 88
5.3.2 Core API 89
5.4 Case Studies 92
5.4.1 Big Data management systems 93
5.4.2 P2P systems 94
5.5 Summary 97

6 Load Balancing Implementation and Experiments 99
6.1 Load Balancing Implementation on EventCloud 100
6.1.1 Variations on components 100
6.1.2 Experiments 103
6.2 Experiments on Variable Hash Functions 108
6.2.1 Computing a new hash function 108
6.2.2 Load balancing policies 109
6.2.3 Experiments 114
6.3 Variable Hash Functions on a Chord Ring 125
6.3.1 Concept and specificities 125
6.3.2 Experiments 130
6.4 Summary 132

7 Conclusion 135
7.1 Summary 135
7.2 Perspectives 137

A Version française 141
A.1 Introduction 142
A.1.1 Motivation et Problématique 142
A.1.2 Plan et Contributions 144
A.2 Résumé du contenu de la thèse 146
A.2.1 Fonctions de Hachage Variables 146
A.2.2 API Générique pour l'Equilibrage de Charge 147
A.2.3 Implémentation et Expériences 149
A.3 Conclusion 151
A.3.1 Résumé 151

A.3.2 Perspectives 153

List of Acronyms **167**

List of Figures

2.1 Presentation vs. Semantics. 9
2.2 Example of an RDF graph. 11
2.3 Example of a 2-dimensional CAN. 15
2.4 Node arrival process in a CAN overlay. 15
2.5 Node departure process in a CAN overlay. 16
2.6 Example of a Chord overlay. 18
2.7 EC architecture. 20
2.8 Peers potentially matching with a subscription. 23

3.1 Taxonomy of load balancing solutions in P2P systems. 37
3.2 HotRoD architecture. 43
3.3 NIXMIG approach proposed by Konstantinou et al. 47

4.1 Standard hash function of a CAN. 58
4.2 Default hash function inefficient to disseminate data. 59
4.3 Enlargement of the area dedicated to Latin data. 61
4.4 Hash function associated with an enlarged area for skewed values. . . 62
4.5 Successive updates on hash functions. 63
4.6 Consequences of not spreading the new Unicode value of a bound. . . 66
4.7 Hash function update message routing. 67
4.8 Unicode reduction process on the C_{max} bound. 68
4.9 Inconsistent reductions for the same CAN-based value (0.5). 72
4.10 Bound reduction on peers responsible for two Unicode intervals. . . 73
4.11 Propagation of a new Unicode value and consequences. 75
4.12 Overall hash function knowledge of a peer. 76

4.13 Possible routing path associated to an item lookup. 79

5.1 Basic abstractions for a generic API. 88
5.2 Workflow associated with the second scheme proposed by Rao et al. 95

6.1 Middle vs. Centroid split strategies. 104
6.2 Load balancing partitioning using middle vs centroid. 105
6.3 Middle split strategy unable to move data. 106
6.4 Load distribution before and after rebalancing. 110
6.5 Threshold-based limit strategy. 113
6.6 Load distribution depending on the strategy used. 119
6.7 Load distribution (excluding peers storing no data) 121
6.8 Variable hash functions on a Chord overlay. 128

7.1 Possible RDF stream processing architecture. 139

List of Listings

2.1 RDF triple example. 10
2.2 RDF triple example. 10
2.3 Basic SPARQL query example. 11
2.4 SPARQL query using the FILTER clause. 12
2.5 Subscription example. 20
2.6 Event pattern example. 20

List of Tables

2.1 Unicode character ranges associated to some alphabets/symbols. . . 22

5.1 Load balancing strategies for Big Data systems. 86
5.2 Load balancing strategies for P2P systems. 87

6.1 Load balancing strategies comparison in the EC. 107
6.2 Data dissemination over peers. 118
6.3 Data distribution when using various implementations. 124
6.4 Load balancing results for each strategy applied on a Chord ring. . . 131

Chapter 1

Introduction

Contents

1.1 Motivation and Problem Definition 1
1.2 Outline and Contributions 3

1.1 Motivation and Problem Definition

Many applications need to integrate data at Web scale to extract information and knowledge. The use of Web data can be related to social networking, media sharing or even business intelligence. Among the new Web 3.0 technologies, the Semantic Web brings meaning to every Web element and provides useful tools for describing knowledge and reasoning on Web data. As the Internet is growing exponentially, more and more data is generated every day, which led to the notion of Big Data, referring to large collections of heterogeneous data produced by various sources. As a result, such data is usually loosely structured and hence the differences of size, popularity or content between resources can vary substantially. For this reason, real world datasets, including those produced on the Web, are known to potentially contain highly skewed values.

With the advent of Big Data, it becomes incredibly difficult to manage realistic datasets on a single machine. To face the incredible amount of information to man-

age and capitalize on it, while providing sufficient performance to users in various contexts, many distributed solutions are available, including Peer-to-Peer (P2P) systems, distributed Not only SQL (NoSQL) databases, Cloud computing services or stream processing engines. All these systems represent efficient and scalable solutions for data storage and processing in large distributed environments. The work behind this thesis was strongly motivated by the building in our research team of a P2P-based system for Semantic Web data storage and retrieval, geared towards situational-driven adaptability, taking the form of an event marketplace platform.

This thesis covers some challenges associated with the dissemination of biased data among nodes, which is a growing load imbalance problem for distributed systems. With the emergence of the Semantic Web and Big Data, the storage of items in an order-preserving way has become very frequent to retrieve sets of close values faster from contiguous nodes, which limits the number of hops when solving a query. Consequently, the same subset of nodes is systematically contacted for incoming data/queries, which may overload these nodes. Based on the observations made on our distributed platform, using such architecture and facing this problem of load imbalance, we propose in this thesis a novel approach to improve data distribution among nodes in Structured Overlay Networks (SONs). Our technique preserves the ordering of data and does not require node migration, nor data replication, which are common but costly solutions regarding the update of the overlay topology or the consistency constraints on data. Our contribution aims at dynamically managing load imbalance by allowing different peers to use different hash functions, while maintaining consistency of the overlay. In SONs, the hash function applied by peers to insert or retrieve an item is at the heart of data distribution. However, few load balancing solutions rely on multiple hash functions, and usually these strategies aim at replicating popular data. In our approach, we allow a peer to change its hash function to reduce its load. Since this can be done at runtime, without a priori knowledge regarding data distribution, this provides an efficient and adaptive load balancing mechanism.

Load balancing is a key factor for any distributed system, in particular for

systems geared towards data dissemination. Imbalances are usually caused by an unfair partitioning of keys/network identifiers, frequent node arrival and departure or heterogeneity in terms of bandwidth, storage and processing capacities between nodes. As the produced data needs increasingly to be analysed/manipulated/retrieved in real time by a potentially large number of users, addressing load imbalance is necessary, especially to minimize response time and avoid workload being handled by only one or few nodes. To do so, there exist almost as many load balancing strategies as there are different systems. Besides, the proposed solutions are often coupled to their own API, making it difficult to port a scheme from a system to another. Yet when developing a distributed system, we believe it would be useful to be able to try on different solutions, as it is often not so easy to anticipate which strategy would be the most efficient and suitable for a particular system. In this thesis, we propose to build a common API to implement any load balancing strategy independently from the rest of the code. We show that many load balancing schemes are comprised of the same basic elements, and only the implementation and interconnection of these elements vary. By properly defining these elements and their behavior, an unlimited number of possible load balancing strategies may be conceived and implemented.

1.2 Outline and Contributions

This thesis relates to load balancing in distributed systems for data storage and processing. The contribution is mainly organized into three chapters, which are Chapters 4, 5 and 6. We first present our load balancing strategy using variable hash functions for skewed data dissemination. Then, we propose an API to easily implement any load balancing strategy in a distributed system. Finally, we detail the experiments we performed using this API, in particular to implement our variable hash functions strategy. Overall, this thesis is organized as follows:

- **Chapter 2** gives an overview of the concepts covered by this thesis. We first introduce the notion of Semantic Web, with a focus on how to represent and query data. Then, we present the Peer-to-Peer paradigm and its different types of overlays. Finally, we detail the EventCloud architecture, a Peer-to-

Peer system for Semantic Web data storage and retrieval developed in our research team, which provides the technical basis of this work.

- **Chapter 3** presents a state of the art of load balancing solutions for distributed systems for Big Data storage and processing. We describe the main existing techniques to address imbalances regarding workload distribution among nodes. We especially consider P2P systems for storage and retrieval of skewed/popular values and/or preserving the natural ordering of data, to correlate with the architecture and the issues faced by the EventCloud.

- **Chapter 4** describes the solution we propose to address load imbalances regarding skewed data distribution in P2P systems preserving the natural ordering of data. As hash functions are at the heart of data distribution for Distributed Hash Table (DHT)-based overlays, we propose the use of several hash functions at the same time in a network, to improve skewed data dissemination. Our goal is to allow an overloaded peer to modify the hash function it applies on data, in order to be responsible for a smaller interval of values and hence store less items. This contribution has been presented at the PDCAT conference in 2014 [1].

- **Chapter 5** presents a generic API to implement most existing load balancing strategies for distributed data storage systems. We show how any strategy can be decomposed into criteria and how changing the behavior of these criteria allows to create an unlimited number of different strategies. The API is useful to switch from a strategy to another in a few lines of code, which can be helpful when developing a system, to easily experiment various behaviors. This work was first presented as part of the SBAC-PAD 2014 workshops [2]. A revised and expanded version was then published as a special issue paper in Wiley's *Concurrency and Computation: Practice and Experience* journal in 2015 [3].

- **Chapter 6** presents the experiments we performed using our generic API. We show how load balancing was implemented on the EventCloud, as well as the various strategies we tested before eventually choosing the most efficient one to disseminate Semantic Web data among peers. Then, we present

our simulated experiments in a CAN and a Chord overlay to improve data distribution using variable hash functions.

- **Chapter 7** concludes this thesis. It reviews the presented contributions and opens the perspective for some new research initiatives resulting from this work.

Chapter 2

Background: Distributed Systems for RDF Data Storage and Retrieval

Contents

2.1 Semantic Web 8

2.1.1 RDF data model 9

2.1.2 SPARQL query language 11

2.2 Peer-to-Peer Systems 13

2.2.1 Overview 13

2.2.2 CAN 14

2.2.3 Chord 17

2.3 The EventCloud RDF Store 19

2.3.1 Architecture 19

2.3.2 Unicode 21

2.3.3 Subscription matching 22

2.3.4 Optimal broadcast algorithm 23

2.4 Other Existing RDF Storage Systems 24

2.4.1 Centralized approaches 26

2.4.2 Distributed approaches 26
2.5 Load Balancing Challenge 28

Over the recent years, our research group has developed the EventCloud (EC)[1], that can be seen as a distributed database for Semantic Web data on an underlying P2P structure, allowing the continuous injection of large amounts of events and their retrieval. The EC development was originally motivated by the PLAY project[2] which aimed at designing a platform that allows for "event-driven interaction in large highly distributed and heterogeneous service systems". The purpose of this thesis was strongly influenced by the work carried out on the EC.

In this chapter, we first introduce a number of concepts essential to proper understanding of this thesis. Then, we present the EC architecture. Further details about the EC can be found in Laurent PELLEGRINO's thesis [5]. Finally, we review the existing solutions to store and retrieve Semantic Web data.

2.1 Semantic Web

The Semantic Web refers to the W3C's vision of the Web of linked data[3], providing machine-understandable information in order to build a kind of global Web database. As such, the Semantic Web may be seen as a metadata provider to Big Data, as it is meant to add a meaning to large collections of Web resources. Figure 2.1, taken from [6], presents a concrete example of how powerful the Semantic Web is. The goal of such concept is to add a meaning (semantics) to every element that can be found on the Web, and correlate these elements based on their semantics. Ultimately, this is meant to provide new reasoning capacities on these elements. To achieve this, the Semantic Web relies on several well-known Web standards and technologies. As Web data is mostly made of text and these resources are accessed by links, the Semantic Web data is represented by Unicode

[1] "Event" because, ultimately, data is meant to represent events, and "Cloud" as the system can be deployed on a Cloud infrastructure. The EventCloud has been deposited at the APP (Agence pour la Protection des Programmes) [4]
[2] `http://www.play-project.eu`
[3] `http://www.w3.org/standards/semanticweb/`

characters and frequently makes use of IRIs[4] pointing to these Web resources.

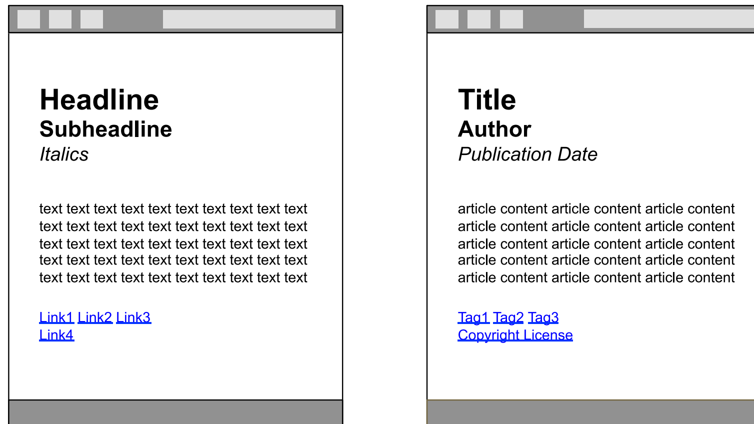


Figure 2.1 – Presentation vs. Semantics (taken from [6]).

2.1.1 RDF data model

The Resource Description Framework (RDF) [7] is a W3C technology used to represent machine-processable semantic data. RDF provides a powerful abstract data model for structured knowledge representation and is used to describe semantic relationships between data. Resources are represented as *triples* in the form of $< subject, predicate, object >$ expressions. The *subject* of a triple indicates the resource that the statement is about, the *predicate* defines a property of the *subject*, and the *object* is the value of the property. An example of a basic RDF triple is shown in Listing 2.1. It represents a resource about the city of Vienna, with which an abstract is associated. The subject and predicate values are in the form of IRIs because they refer to online resources. These IRIs are made of a prefix, representing the namespace to which the values belong. This triple is taken from one of the DBpedia [8] datasets available online[5]. DBpedia is one of the most popular provider of Semantic Web data to date. It is the semantic version of the famous online encyclopedia Wikipedia, and provides datasets containing extracted

[4]Internationalized Resource Identifier (IRI) is an extension of Uniform Resource Identifier (URI), created in order to represent any Unicode character, unlike traditional URIs that are restricted to the ASCII characters.

[5]`http://dbpedia.org/`

information from Wikipedia, to which is added a layer of semantics, offering new reasoning possibilities on such data. Nowadays, more and more organizations extract information from DBpedia to provide their users a new experience when searching for information. A non-exhaustive list of such organizations includes the BBC [9], New York Times[6] and Thomson Reuters[7].

```
PREFIX dbpedia:   <http://dbpedia.org/resource/>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
TRIPLE: (dbpedia:Vienna, rdfs:abstract, "Vienna is the capital...")
```

Listing 2.1 – RDF triple example taken from a DBpedia dataset.

The most interesting aspect of RDF lies in its linking structure. Predicate values act as links between resources, and these links bring semantics to data. When processing the two triples of Listing 2.2, a machine should be able to understand that *Helen* and *Elsie* are sisters and form a family along with *John* (i.e. semantic relations). To help machines capture the meaning of a family and the links between siblings, the structure of a family must have been defined beforehand. In the Semantic Web world, this is done in a so-called ontology. An ontology represents a set of knowledge about a particular domain (e.g., medicine, cinema, linguistics) and precises, among others, the interrelationships that exist between things in this domain. Thus, very often, a predicate is defined in an ontology to add a meaning (meta-data) to its value. The most common standards to design an ontology are RDF Schema (RDFS) and Web Ontology Language (OWL).

```
TRIPLE 1: (Helen, hasFather, John)
TRIPLE 2: (Elsie, hasFather, John)
```

Listing 2.2 – RDF triple example taken from a DBpedia dataset.

When connected together, triples form a directed graph where arcs are always directed from resources (subjects) to values (objects). The label associated to an arc refers to the predicate value of the triple. For this reason, there exist fewer distinct values of predicates than for the other parts of a triple, as a single predicate

[6] http://data.nytimes.com/
[7] http://www.opencalais.com/

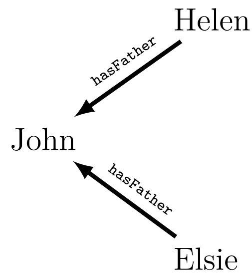


Figure 2.2 – Example of an RDF graph for the triples of Listing 2.2.

value may be associated n times to n different resources. This low diversity among terms makes RDF data very skewed. To support this claim, the authors of [10] have shown a DBpedia dataset may contain the same popular term in up to 55% of its triples.
A graph built using the triples of Listing 2.2 is depicted in Figure 2.2. Such graphs can potentially extend massively, like social graphs, for instance. A graph may be identified by a value (an IRI) to provide more information on data provenance. In this case, the graph is called a *named graph* [11]. This explains why the notion of *triple* is sometimes replaced by that of *quadruple*, that is composed of a triple to which a graph value is associated.

2.1.2 SPARQL query language

```
1 PREFIX dc: <http://purl.org/dc/terms/>
2 SELECT ?author
3 WHERE {
4     GRAPH ?g {
5         ?book dc:subject dc:Science .
6         ?book dc:creator ?author
7               }
8       }
```

Listing 2.3 – Basic SPARQL query example retrieving all authors of a scientific book.

SPARQL Protocol and RDF Query Language (SPARQL) [12] is a W3C recommendation to query RDF data. This query language is based on the concept of

triple patterns. A triple pattern refers to triple whose values are either constant or variable. Two triple patterns may be linked with each other if they share a common variable. Conjunctions can be performed on triple patterns sharing a common variable in order to join values matching each independent triple pattern. To support quadruples, the notion of *triple pattern* can be extended to that of *quadruple pattern* in order to query for triples present in the same graph or not.

An example of a *join query* is presented in Listing 2.3. The SPARQL syntax is similar to that of SQL: the SELECT clause returns the variables to retrieve. It is worth mentioning that, in SPARQL, all variables are identified by the "?" character preceding the variable name. The WHERE clause contains all the triple patterns to match against RDF data. In the present case, two triple patterns are joined using the "." character, hence they are linked together on their common variable ?*book*. The *dc* prefix of the constants refer to the Dublin Core [13] namespace, that provides an ontology for describing resources such as publications. Finally, the GRAPH clause identifies the graph to which the values must belong. In our case, they can belong to any graph as a variable (?*g*) is used to identify the graph. Therefore, the query of Listing 2.3 returns all authors of a scientific book.

```
1  PREFIX dc: <http://purl.org/dc/terms/>
2  SELECT ?author
3  WHERE {
4      GRAPH ?g {
5          ?book dc:subject dc:Science .
6          ?book dc:creator ?author .
7          ?book dc:dateCopyrighted ?date .
8          FILTER (?date >= 2012 && ?date < 2016)
9               }
10       }
```

Listing 2.4 – SPARQL query using the FILTER clause to retrieve the authors of scientific books copyrighted between 2012 and 2015.

To add more constraints on the results, a FILTER clause may be added to the WHERE clause. It is used to refine the results of a query in order to return

only those that validate the constraints defined in the FILTER clause. Usually, filtering is done on text, numbers or dates. An example of a SPARQL query using the FILTER clause is presented in Listing 2.4. It sets a specific condition for the authors of scientific books: only those that wrote a book copyrighted between 2012 and 2015 will be included in the results.

2.2 Peer-to-Peer Systems

2.2.1 Overview

A P2P system refers to a distributed architecture for applications like file sharing (e.g., BitTorrent [14]), data storage (e.g., Dynamo [15]) or distributed computing (e.g., SETI@home [16]). Unlike the client/server model, each machine (also denoted as peer or node) can be both client and server. The network is made of interconnected nodes sharing their resources with each other. Among the main benefits of P2P, we can highlight the scalability (new peers can be added to the network to adapt to demand), fault-tolerance (should one peer fail, another one will take over in a transparent way for the user) and a full decentralization (the system can run without any central coordination). P2P overlays are usually classified into three main categories: *unstructured*, *structured* or *hierarchical* overlays.

Unstructured overlays In these systems, there is no constraint regarding the architecture. The overlay is built with peers joining at random locations, thus creating random links between peers. This offers robustness to frequent peer arrival and departure (also known as *churn*). Therefore, this is a very simple technique to build a P2P overlay. However, data is also arbitrarily placed on peers, which leads to a more complicated solution when searching for data. The most common technique consists in flooding the whole network to lookup a resource, which results in increased overhead, especially when searching for a very rare resource. Many protocols for file sharing were based on unstructured overlays, like BitTorrent [14] and Gnutella [17].

Structured overlays In order to address the shortcomings of unstructured overlays, a new type of overlay was developed: the Structured Overlay Network (SON). Many P2P systems are based on a SON, like Chord [18], CAN [19], Kademlia [20] and Pastry [21]. Such overlays are organized into a geometrical topology (e.g., cube, ring) and usually implement a DHT to distribute and locate data. Each node is assigned a set of identifiers from a common identifier space. A DHT maps data into the identifier space by using a consistent hash function [22] (usually SHA1 or MD5): each data item is associated with a key, and hashing this key provides the coordinate in the identifier space to which this item belongs. Usually, in an overlay, all peers apply the same uniform hash function to place and locate data. Thus, a peer can more easily locate a resource than with unstructured overlays. To route a lookup query, peers use information they maintain about other peers in the overlay, like their neighbors (the peers they are linked with). This allows to build efficient routing strategies in order to minimize the number of hops to reach the desired resource. However, this implies that peers always maintain a consistent view of the network which may increase the overhead in case of a high churn rate.

Hierarchical overlays These systems exploit properties of both unstructured and structured overlays. Peers are organized into groups and these groups are organized into overlays. A hierarchical architecture is made of several overlays interconnected into a tree-like structure. The different overlays in this hierarchy can use different routing mechanisms. Examples of hierarchical overlays can be found in [23], [24] and [25].

In the following subsections, we introduce two types of structured overlays: CAN and Chord.

2.2.2 CAN

A Content Addressable Network (CAN) network is a decentralized and structured P2P infrastructure that can be represented as a *d-dimensional* coordinate space containing n peers. Each peer is responsible for the zone it holds in the network

(a set of intervals in this space). All dimensions have a minimum and a maximum CAN-based value C_{min} and C_{max} (0 and 1 in Figure 2.3). Each peer p owns an interval $[min_p^d; max_p^d[$ delimited by 2 bounds (lower and upper bounds) between C_{min} and C_{max} on each dimension d. A CAN-based interval is fixed and can only be modified during *join* or *leave* node operations.

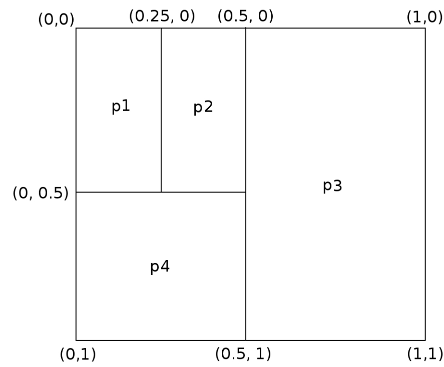
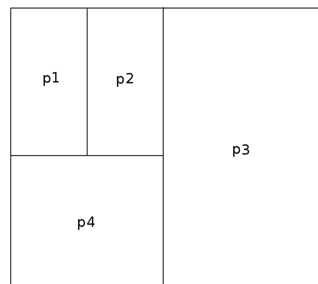
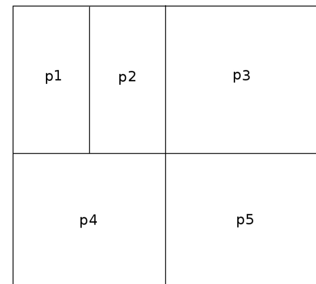


Figure 2.3 – Example of a 2-dimensional CAN.



(a) Overlay made of 4 peers.

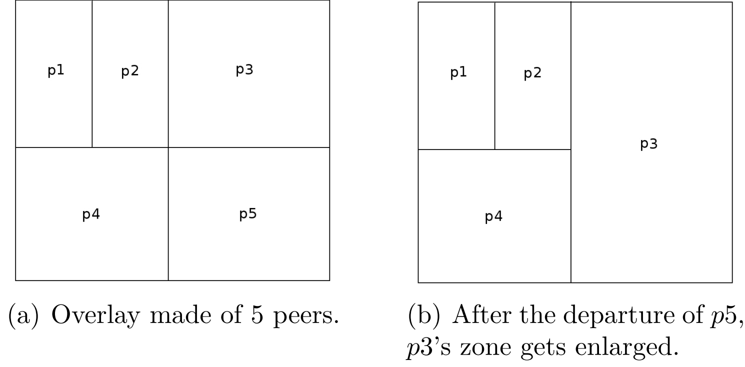


(b) Arrival of a fifth peer that splits with $p3$.

Figure 2.4 – Node arrival process in a CAN overlay.

Node arrival The *join* procedure consists in adding a new peer p_{new} in the network by splitting in two even surfaces z_1 and z_2 the randomly chosen zone z of an existing peer p_{old}. Half of z remains managed by p_{old} in z_1, while the other half (z_2) becomes p_{new}'s zone. As a result, all data items that were included in z_2 are now stored by p_{new}. This *join* operation also affects the network topology, as the

peers holding the zones adjacent to z_2 are no longer neighbors of p_{old} but become the neighbors of p_{new}. For more clarity, the node arrival process in a CAN overlay is depicted in Figure 2.4.



(a) Overlay made of 5 peers. (b) After the departure of $p5$, $p3$'s zone gets enlarged.

Figure 2.5 – Node departure process in a CAN overlay.

Node departure Conversely, the *leave* procedure consists in merging the zone z_{leave} of the leaving peer p_{leave} with the zone z_{chosen} of one of its neighbors p_{chosen}. If z_{leave} and z_{chosen} can be merged to create a new valid CAN zone, then p_{chosen} is definitely selected and both peers start merging. Otherwise, if merging with this chosen neighbor does not work, p_{leave} picks its neighbor with the smallest zone as the new p_{chosen}. Finally, p_{chosen} hands over p_{leave}'s data items along with p_{leave}'s neighbors on z_{leave}. The node departure process is depicted in Figure 2.5.

Message routing Each peer can only communicate with its neighbors, thus routing from neighbor to neighbor has to be done in order to reach remote zones in the network. The CAN topology is a torus which means, in Figure 2.3, that peers $p1$ and $p3$ are neighbors on the horizontal dimension. CAN provides a DHT abstraction, with resources being indexed on keys. An item it, in a d-dimensional CAN, is associated with a key that is a set of d coordinates (one for each dimension) obtained by applying d uniform hash functions on it. The obtained coordinates correspond to a point in the CAN coordinate space. Thus, when a peer receives a new data item to insert or a query to execute, it has to hash it to check whether it is responsible for it or not. If the hash value does not

match the peer's CAN coordinates, it means the peer is not responsible for the corresponding item. In this case, the peer chooses a dimension and forwards the item/query on the same dimension until it reaches the correct coordinate for this dimension, then on another dimension and so on until the message reaches the right peer.

2.2.3 Chord

In Chord, nodes are organized in a ring topology. Each node is identified by its *node identifier* (obtained by hashing its IP address) and all nodes are ordered in the circular identifier space according to their node identifier value. The neighbors of a node are its predecessor and successor in the ring.
A *key identifier* is associated to each node and each item. By default, a node's key identifier corresponds to its node identifier under the consistent hash function used in the overlay. A node is responsible for storing items whose key identifier falls between its predecessor's key identifier (excluded) and its own key identifier (included). Thus, a key k is stored at the first node after k in the ring order, i.e. $successor(k)$. For example, in Figure 2.6, key k_{59} is located at node n_{60}. Chord uses a consistent hash function to distribute key identifiers on peers, that is supposed to naturally balance the load evenly among nodes as each peer should approximately be responsible for the same number of keys.

Message routing In Chord, routing is done in a clockwise direction, i.e. from a peer to its successor. A node n that receives a lookup query for a key k has to evaluate whether k is included in $]key_n; key_{n.successor}]$ or not. If so, it means that n's successor is responsible for k and routing will be stopped when reaching $n.successor$. Otherwise, the same process will be executed by $n.successor$ and so forth until the peer managing k is found. In order to accelerate this routing process, nodes also maintain links in a routing table called *finger table* that contains the identity (node/key identifier, IP address) of other peers, located possibly far away in the overlay. Thus, when receiving a lookup for k, if n's successor is not responsible for k, n can pick an entry in its finger table that is closer to k in

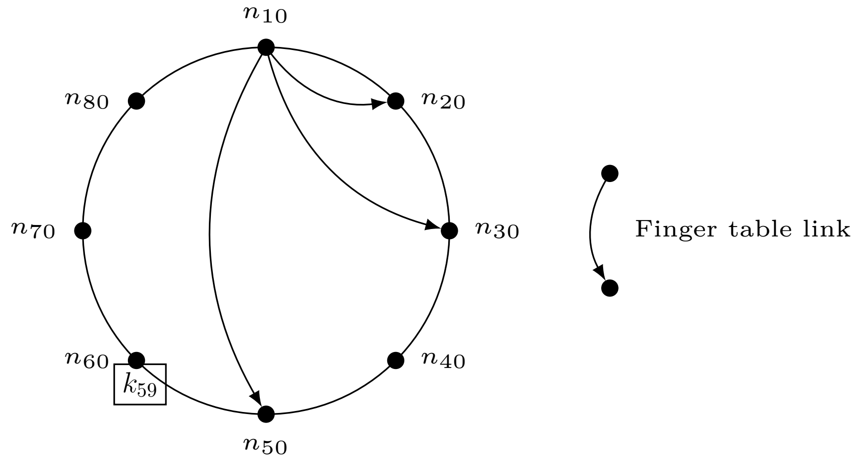


Figure 2.6 – Example of a Chord overlay.

the identifier space than n's successor. The message will be routed to the node corresponding to this finger table entry instead of being routed to n's successor. For instance, in Figure 2.6, if node n_{10} is looking for key k_{59} (stored by n_{60}), it can send *lookup*(k_{59}) directly to n_{50}, because n_{50} is closer to k_{59} in terms of hops than n_{10}'s successor n_{20}.

Node joins and departures Let us consider two existing nodes n_{10} and n_{30}, with n_{10} being the predecessor node of n_{30}. When a new node n_{20} arrives in the overlay between these two peers, n_{20} becomes the new successor of n_{10} and the new predecessor of n_{30}. Thus, n_{20} becomes responsible for all the keys comprised between $]n_{10}; n_{20}]$. This implies that n_{20} acquires n_{30}'s items whose key is included in this interval. The reverse process would be applied if n_{20} decided to leave the network: the keys and data managed by n_{20} would be acquired by its successor n_{30}. In order to maintain up-to-date information about their neighbors and finger table entries, all nodes periodically run a *stabilization* protocol. This protocol verifies the correctness of a peer's links and allows a peer to update its knowledge if necessary, for example after a node has arrived or left.

2.3 The EventCloud RDF Store

The EC is a Java software platform that offers the possibility for services to communicate in an asynchronous and loosely coupled fashion thanks to the publish-subscribe paradigm but also to store and to retrieve past events in a synchronous manner. The EC enables publish/subscribe interactions by means of its event-driven architecture. Subscribers register their interest in some types of events in order to asynchronously receive the ones that match their concerns. Events are semantically described as a list of quadruples. Quadruples are in the form of $< graph, subject, predicate, object >$ tuples where each element is named an RDF term in the RDF [7] terminology. All quadruples included in the same event are closely related and share the same graph value that identifies the event. Alternatively, a synchronous mode allows to query for past events that were published earlier and are now stored in the system. The underlying EC architecture is based on a slightly modified version of the CAN [19] P2P infrastructure as described next.

2.3.1 Architecture

An EC is defined with four dimensions to map each RDF term of a quadruple to a dimension of the P2P network. The first dimension is associated to the *graph* value, the second dimension to the *subject* value, and so on. The overall EC architecture is depicted in Figure 2.7. Also, in contrary to the default CAN protocol that makes use of consistent hashing to map data onto nodes, the EC uses the lexicographic order to index data. Thus, a quadruple directly matches a point in a 4-dimensional coordinate space. The lexicographic order helps cluster items sharing similar values from a syntactic point of view, which often means that these values are semantically close as well (e.g., if they share a common prefix). As a result, range queries, for instance, can be resolved with a minimum number of hops. In this way, as we consider an event is made of lexicographically close data, we can expect all of an event's quadruples to be stored by the same peer or at least in contiguous peers in the CAN. As we consider that a user's query/subscription is likely to include a set of related information, the use of the lexicographic order

simplifies range queries execution (e.g., when querying for a range of temperatures) and data retrieval (e.g., content related to a city, a football match or even a friend from a social network) by limiting communication between peers when solving a query or matching a publication against a subscription.

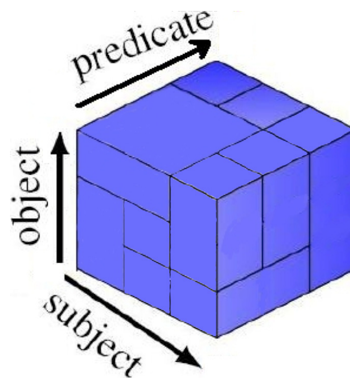


Figure 2.7 – EC architecture. For clarity purposes, the *graph* value axis has been occulted.

```
1 PREFIX foot : <http://example.org/football_matches/>
2 SELECT ?event WHERE {
3 GRAPH ?g {
4     foot:Barcelona_vs_Bayern ?occurs ?event
5     }
6 }
```

Listing 2.5 – Example of a subscription notifying its subscriber about any event occuring during the match Barcelona vs. Bayern Munich.

```
EVENT football_graph {
        foot:Barcelona_vs_Bayern foot:goalScored foot:goal1234 .
        foot:goal1234 foot:scorer player:Messi .
        foot:Barcelona_vs_Bayern foot:currentScore ‘‘2-0’’
}
```

Listing 2.6 – Event pattern example.

Listing 2.5 presents an example of subscription to retrieve information about any event (e.g., goals, yellow cards, substitutions) that takes place during the football match Barcelona against Bayern Munich. A subscription takes the form of a SPARQL query made of one or more quadruple patterns (in the present example there is one). An event is made of several quadruples (sharing the same graph value). An example of event for a goal scored by Messi during this match (*Barcelona_vs_Bayern*) is shown in Listing 2.6. The main advantage of using the lexicographic order is that all of the triples that make up this event will necessarily share identical values on some axis. Obviously, the *graph* axis is concerned as there is only one value for the whole event, but also potentially other axis such as the *subject* in Listing 2.6 (*Barcelona_vs_Bayern* is shared by two quadruples). When using a randomizing hash function, these triples would have been scattered across the network, making it more costly to bring them all together to answer a query/subscription.

2.3.2 Unicode

To support the lexicographic order in the EC, and as RDF data is made of Unicode characters, each CAN bound in the EC is associated with Unicode characters. This allows to define the Unicode values that can be stored between two bounds. A Unicode character is encoded as a 32-bit integer, also known as codepoint, whose value may range from 0 to $10FFFF_{16} = 1114111$. To date, the Unicode table of characters includes nearly all of the world's alphabets, along with symbols like mathematical operators or emoticons. The interested reader can find more information on these characters on the Unicode Consortium website[8].
Table 2.1 provides an overview of the codepoint ranges associated with some of the most used characters worldwide. Latin characters are the first in the Unicode table of characters. For instance, the character "*A*" corresponds to the codepoint value 65. The basic ASCII subset includes about 100 characters, while East Asian characters (Chinese, Japanese, Korean), located farther in the table, occupy a much wider range, representing more than 70000 characters when including extensions[9].

[8] http://unicode.org/

[9] An extension refers to a range of codepoints in the Unicode table that usually contains uncommon characters only used in some regions of the world or in the past centuries.

The fact that Latin characters represent such a small interval of values among the one million characters of the Unicode table makes the distribution of Latin data very skewed when using the lexicographic order. To decrease this impact on distribution, the EC allows to define at startup the lower and upper Unicode bounds of the CAN space. Hence, one may exclude non-Latin characters if the data to be stored is known beforehand.

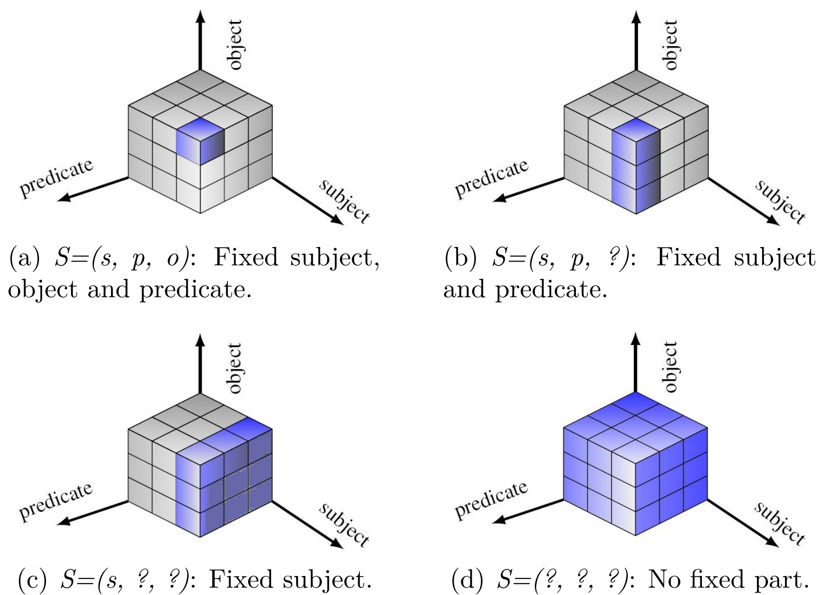
Script	**Range**	**Number of characters**
Basic Latin (ASCII)	0000–007F	128
Greek	0370–03FF	144
Cyrillic	0400–04FF	256
Thai	0E00–0E7F	128
Mathematical Operators	2200–22FF	256
Emoticons	1F600–1F64F	80
CJK (Chinese, Japanese, Korean) extension B	20000–2A6D6	42711

Table 2.1 – Unicode character ranges associated to some alphabets/symbols.

2.3.3 Subscription matching

For more clarity, we will consider a simplified subscription S that may be represented as a single quadruple pattern with some wildcards denoted by "*?*", such as *S=(g, s, p, ?)*. Indexing this subscription in the EC simply consists in sending S to all the peers that manage the fixed attributes g, s and p on the first three dimensions. Unlike some other RDF stores (see Section 2.4.2), publications (i.e. events) are indexed only once. However, a subscription that contains wildcard(s) is indexed on several peers as all the peers on the wildcard(s) axis may potentially match this subscription. Consequently, the number of peers concerned by a subscription depends, in part, on the number of fixed attributes of the subscription: the fewer fixed attributes, the more concerned peers there are.
The propagation of various subscriptions is shown in Figure 2.8. For example, in

Figure 2.8(c), the subscription *S=(s, ?, ?)* aims at retrieving all data concerning the specified subject s. To index this subscription on peers that may store such data, S must be sent to peers whose bounds on the subject dimension include s. As no particular value is specified for the predicate and object, the subscription could match with any value and hence there is no constraint regarding the predicate and object intervals of peers. Therefore, many peers may store this subscription as it will be propagated on the whole predicate and object dimensions. The same principles apply when synchronously querying for past events stored by one or more peers [26]. In the standard implementation, each peer owns a Jena TDB [27] component that acts as an RDF storage engine responsible for executing queries and indexing data.



(a) *S=(s, p, o)*: Fixed subject, object and predicate.

(b) *S=(s, p, ?)*: Fixed subject and predicate.

(c) *S=(s, ?, ?)*: Fixed subject.

(d) *S=(?, ?, ?)*: No fixed part.

Figure 2.8 – Peers potentially matching with a subscription S, depending on the constant parts in S (taken from [26]).

2.3.4 Optimal broadcast algorithm

To solve queries like those presented in Figure 2.8, the initial peer sending a query cannot know exactly how many peers potentially match the query criteria. Thus, to propagate a query, a naive approach consists in using message flooding through

each peer's neighbors, which results in duplicate messages sent to peers. Although such messages are ignored, their routing consumes bandwidth and they may overload peers processing them. In order to reduce the number of messages propagated through the whole network to an absolute minimum, an optimal broadcast algorithm for CAN [28] has been implemented in the EC. This algorithm is said to be optimal because a peer receives a broadcasted message once and only once. In the EC, this algorithm is also used for other purposes: for example, each peer periodically broadcasts information about its load state, which is useful when performing a load balancing operation.

Algorithm 2.1 presents a simplified version of the optimal broadcast algorithm. The behavior of this algorithm can be summarized as follows: when a peer wants to broadcast a message, this peer selects all its neighbors and forwards them the message if they validate two main constraints (lines 14 and 22). A message is routed on a given direction in the CAN (backwards or forwards, regarding the sender's location). Initially, the routing process occurs on all dimensions and both directions. Recipients continue routing the message following the same principle, until none of their neighbors can validate the constraints or C_{min}/C_{max} has been reached on a routing dimension. If so, the recipient continues routing only on the lower dimension(s), until C_{min}/C_{max} has been reached on dimension 0 or no neighbor matches the constraints.

2.4 Other Existing RDF Storage Systems

As the Semantic Web is growing in popularity, there exist more and more RDF storage systems, consisting in either centralized or distributed repositories. A non-exhaustive list of existing solutions includes Relational Database Management Systems (RDBMS), NoSQL stores, structured or unstructured P2P systems. In the following, we briefly describe the main concepts behind each of these solutions.

1: **function** ROUTE__UPDATE__BOUND__MESSAGE($msg, dim, direction$)
2: **if** $direction = forwards$ & $max_{peer}^{dim} = C_{max}$ **then**
3: stop routing
4: **else if** $direction = backwards$ & $min_{peer}^{dim} = C_{min}$ **then**
5: stop routing
6: **else**
7: **for each** $neighbor \in peer$.getNeighbors(dim,$direction$) **do**
8: **if** CHECK__CORNER__CONSTRAINT($neighbor, msg$) & CHECK__SPATIAL__CONSTRAINT($neighbor, msg$) **then**
9: route(msg, $neighbor$)
10: **end if**
11: **end for each**
12: **end if**
13: **end function**

14: **function** CHECK__CORNER__CONSTRAINT($neighbor$, msg): Boolean
15: **for all** $dim > msg.current_routing_dimension$ **do**
16: **if** $min_{neighbor}^{dim} < min_{peer}^{dim}$ **then**
17: **return** $false$
18: **end if**
19: **end for**
20: **return** $true$
21: **end function**

22: **function** CHECK__SPATIAL__CONSTRAINT($neighbor$, msg): Boolean
23: **for all** $dim < msg.current_routing_dimension$ **do**
24: **if** !($min_{neighbor}^{dim} \leq min_{peer}^{dim} \leq max_{neighbor}^{dim}$) **then**
25: **return** $false$
26: **end if**
27: **end for**
28: **return** $true$
29: **end function**

Algorithm 2.1 – Default optimal broadcast algorithm. In this algorithm, *peer* refers to the current peer processing the the message.

2.4.1 Centralized approaches

The first solutions for RDF data storage, like Oracle RDF Match [29], Sesame [30] and 3store [31], consisted in converting triples into tuples in order to store them like any other record in an RDBMS. Using this approach, the tables can take different forms. A table can be made of an identity column along with three columns corresponding to each part of a triple. Alternatively, the suject value can be used as the key column of the triple with the other parts corresponding to normal columns. Finally, another approach uses vertical partitioning [32], which corresponds to creating a table for each distinct predicate (as they should be few in number). These tables are made of two columns storing respectively subject and object values. However, the structure of traditional RDBMS was created a long time ago and these systems were not designed to store RDF data. They have no notion of semantics, which may complicate the execution of queries based on an RDF graph model (i.e. how to convert a SPARQL query into SQL). This is why specific databases called native RDF stores, like Jena TDB [27], were created to store and query RDF data. These native RDF stores are similar to RDBMS but use an optimized structure for managing triples and support SPARQL, which should provide better performance regarding their query processing mechanism. However, storing RDF triples generally requires to manage large volumes of data, thus scalability. This is why native RDF stores tend to be deployed on distributed architectures, like the EC that uses a Jena TDB instance on each peer.

2.4.2 Distributed approaches

With the increasing amount of data to process, distributed solutions become more and more popular as they provide scalability and are more reliable than centralized solutions suffering from bottlenecks and a single point of failure.

NoSQL Over the last years has emerged the NoSQL movement. NoSQL databases relax some of the relational and transactional constraints specific to RDBMS (the ACID properties) in order to provide better scalability and fast access to store and retrieve data. Solutions based on NoSQL include key/value-oriented stores (e.g., Apache Cassandra [33]), column-oriented stores (e.g., HBase [34]),

document-oriented databases (e.g., MongoDB [35]) and graph-oriented stores (e.g., AllegroGraph [36]). Some NoSQL databases use an underlying P2P structure, like Cassandra, which is fully distributed and uses consistent hashing in a Chord-like topology. In Cassandra, rows can embed several levels of nested key-value pairs. In the overlay, rows are assigned to nodes depending on the hashed value of their key: a row r will be located at $successor(hash(r_{key}))$.
NoSQL databases can be used to store RDF data. For instance, CumulusRDF [37] is a distributed system for RDF data storage implemented on Cassandra. CumulusRDF uses Cassandra's features to index triples. A triple is indexed three times according to three different patterns: SPO (i.e. Subject-Predicate-Object), POS and OSP. This technique is meant to help retrieve triples faster when performing a lookup. For instance, to retrieve data matching the triple pattern $(?s, ?p, o)$, the OSP index should be used as only the object value is defined in the query. For each different index, triples are stored in Cassandra using different nested key-value pairs, for example `{o: {s: {p: -}}}` with the OSP index, where `o` is the key for value `{s: {p: -}}` which is itself a key-value pair, too.

Pure P2P solutions There also exist many RDF stores based on pure P2P solutions. As this thesis mainly focuses on structured P2P overlays, we give below an overview of some existing distributed RDF stores built on top of a structured P2P overlay. An exhaustive review of existing structured P2P storage systems for RDF data can be found in [38]. We will not go into details on RDF stores based on unstructured P2P overlays, like Bibster [39] and S-RDF [40].
Most DHT-based approaches index an RDF triple three times at three different locations by hashing its subject, predicate and object. As all peers use the same hash function to locate data, this eases the retrieval of triples matching a given triple pattern, as presented above for CumulusRDF. Among the existing structured P2P solutions, RDFPeers [41] is a distributed RDF repository built on Multiple Attribute Addressable Network (MAAN) [42], an extension of Chord. In this system, a triple is stored three times at the following locations: $successor(hash(s))$, $successor(hash(p))$ and $successor(hash(o))$. RDFPeers benefits from an efficient range queries resolution thanks to its order-preserving hash function. Another approach, proposed by Battré *et al.* [43], uses a DHT architecture where each node

owns several RDF repositories. The first one acts as a *local* database used by a peer to store triples that originate from it. As in most other papers for this topic, peers also disseminate their local triples at three different locations by separately hashing the three parts of a triple. These disseminated triples, when received by the corresponding peer, are stored in a *received triples* repository. Finally, two other triplestores are maintained by peers: a *replica* database for fault tolerance and a *generated triples* database for reasoning on RDF data.
Very few distributed RDF repositories are implemented on a CAN overlay like the EC. RDFCube [44] is one of the closest systems to the EC, from a topological point of view. RDFCube is built on a 3-dimensional CAN (one dimension for each part of a triple) and its coordinate space is split into *cells* of equal sizes. Each cell contains an *existence-flag*, indicating if a triple is present or not in that cell. However, RDFCube is not meant to store data. It is an indexation scheme of RDFPeers to speed up the processing of join queries over an RDFPeers repository, by reducing the amount of data that has to be transferred between nodes.

2.5 Load Balancing Challenge

Throughout this section, we have presented the notion of Semantic Web along with technologies to represent and query Web data, namely RDF and SPARQL. RDF data is known to be highly skewed, for several reasons. First, as it is meant to represent Web data, most values are in the form of IRIs and hence potentially many triples may share the same prefix. Secondly, the format representing data as a *resource-attribute-value* triple (known as *subject-predicate-object*) necessarily implies the redundant use of identical attributes when describing different resources. Finally, RDF is made of Unicode-encoded values, and whereas the Unicode table of characters contains about one million characters, very often only a very small subset of these characters is found in a dataset.

The EventCloud (EC) that has been developed in our research team is a distributed platform for RDF data storage and retrieval. In the EC, data can take the form of *events*, consisting in several quadruples semantically linked and published together. Based on a CAN overlay, the 4-dimensional architecture of the EC uses

the lexicographic order to place data among peers. This choice was motivated by the fact that a randomizing hash function would destroy the syntactic links between RDF quadruples (for example, the quadruples that compose an event) by randomly placing these items in the overlay. Gathering this scattered data may slow down query execution as the users requests very often imply to retrieve sets of related data. On the contrary, when using an order-preserving hash function, syntactically close values can be stored by contiguous peers. Thus, they can be retrieved in a minimum number of hops, which eases range queries execution, for instance.

Due to this architecture, the EC suffers from an important drawback regarding data dissemination among nodes. It is common that a few peers manage most data items, which may overload these peers regarding their storage and/or processing capacities. Even though RDF data is made of Unicode characters, the size of a single quadruple may be greater than one megabyte (e.g., DBpedia), which will inevitably affect the disk space capacity and more generally the performance of the peer managing such data. Regarding the processing capacity, the few peers responsible for managing a large set of skewed data receive most subscriptions and execute most queries. This necessarily has bad consequences, especially on the overall system performance towards the users sending these queries/subscriptions. These reasons naturally led us to implement load balancing in the EC. In the next chapter, we present a state of the art of existing solutions that relate to the challenges faced by the EC, namely the distribution of skewed values and the preservation of the natural ordering of data.

Chapter 3

Existing Load Balancing Strategies for Distributed Data Storage

Contents

3.1 Load Balancing in non-P2P Systems 33

3.1.1 NoSQL 33

3.1.2 Stream processing 34

3.1.3 Cloud 35

3.2 Load Balancing in Structured P2P Systems 36

3.2.1 Load balancing for RDF data 37

3.2.2 Virtual servers reassignment 39

3.2.3 Data/Node replication 41

3.2.4 Keys reassignment 45

With the advent of Big Data, it becomes incredibly difficult to manage realistic datasets on a single machine. To face the large volume of information to manage and capitalize on it, while providing sufficient performance to users in various contexts, many distributed solutions are available, encompassing P2P systems,

distributed NoSQL databases, cloud computing services or stream processing engines. However, one key issue with all these distributed systems concerns load balancing, in particular for systems geared towards data storage and retrieval. Some nodes may be overloaded compared to others, regarding various load criteria like disk space, CPU or bandwidth consumption. These load imbalances may be caused by one or several factors, including the specifics of a distributed system and its architecture, differences of capacities between nodes, the variation of the popularity of resources over time, and even the nature of the data being stored.

Current distributed systems for Big Data storage and processing exploit real world datasets and, as presented in the previous chapter, these datasets, like Semantic Web data, are known to be highly skewed [10]. The main reason for this lies in the variation of size, popularity and lexicographic similarities among resources. Information that is stored, shared or more generally manipulated can come from different sources, be expressed in various languages (world wide data) and be more or less structured using various formats providing reasoning capabilities, like RDF [7]. A bad dissemination of skewed datasets can quickly create a bottleneck since a biased data distribution can lead to large workloads sent to very few nodes. Also, as more and more data is produced, often to be processed in real time, it becomes crucial to dynamically adapt the allocation of resources depending on the popularity of data (the so-called "trends"), i.e. the frequency of some terms in the produced datasets or the amount of queries generated for a particular resource. These issues typically correspond to the challenges facing the EventCloud to balance skewed Web data (RDF) using non-randomizing hash functions.

More generally, imbalances in distributed systems may also be caused by an unfair partitioning of network identifiers, frequent node arrival and departure or heterogeneity in terms of bandwidth, storage and processing capacities between nodes. To address all these possible load imbalances, many load balancing strategies have been proposed over the last decades, and new strategies still continue to appear in the literature as new types of distributed systems arise. However, regardless of the type of distributed system, most existing strategies usually rely

on similar principles, like node/data relocation, replication, caching or keys reassignment.

In this chapter, we present the main load balancing solutions for distributed storage systems. We first give a brief overview of those used by well-known non-P2P architectures for Big Data storage and/or processing. Then, we discuss existing solutions used by structured P2P systems in more detail, as they are the main topic of this thesis. We especially focus on load balancing strategies used in the context of skewed/popular data storage and/or maintaining data placement in an order-preserving manner, to remain in the context of the EventCloud architecture.

3.1 Load Balancing in non-P2P Systems for Data Management

3.1.1 NoSQL

MongoDB

MongoDB [45] is a document-oriented NoSQL database using sharding (horizontal scaling) to distribute datasets across many machines. Sharding consists in splitting these datasets into chunks to distribute them among several servers (*shards*). Splitting is performed using either range based or hash based partitioning. A chunk has a maximum size value, and whenever it is exceeded, the chunk gets divided into two chunks. Routing instances (*routers*) are responsible for routing queries to the right shard and provide results to the user. These routers use metadata information stored by special instances called *Config Servers* to find the right shard responsible for storing a specific data item. Whenever a shard becomes responsible for a chunk, this information is sent to Config Servers in order to keep the database consistent. Routers also embark a *balancer* process that periodically acquires a lock (so that there is no other router trying to rebalance at the same time) and retrieve from Config Servers the number of chunks stored by each shard. A rebalance will occur only if the heaviest shard stores x (migration threshold) more chunks than the lightest shard. If so, the balancer starts migrating chunks from

the heaviest to the lightest shard until reaching a balanced distribution among the two shards (no more than a difference of one chunk between the number of chunks stored by each shard). Such a rebalance process is likely to occur when a new shard is added or large datasets are inserted. MongoDB users are allowed to configure a few parameters of this strategy: the periodicity of the load balancing actions, the migration threshold and some replication settings to be applied when migrating data.

3.1.2 Stream processing

Stream processing engines, such as Storm [46], process real-time, large-scale data (*streams*) very often continuously produced by sources of data (such as a Twitter data flow). Incoming data is processed by a network of nodes where each node runs an operation on the data it receives, like filtering, before the data is definitely stored. Such mandatorily distributed systems, offering scalability and low latency, are perfectly suited for social-networking applications, whose volume of information is very high and continuously arriving. The main load balancing challenge in stream processing engines consists in ensuring a balanced amount of load (data or tasks) is sent/assigned to each node.

Nasir *et al.*

Nasir *et al.* [47] investigate the direct applicability of the power of two choices paradigm [48] on distributed stream processing systems, focusing on the Apache Storm event processor. The paper considers source instances producing data streams, split into messages (*sub-streams*) sent to workers responsible for consuming and processing these sub-streams. The goal is to address load imbalances in terms of sub-streams managed by each worker. A source that wishes to send a new message of data applies two hash functions $h1$ and $h2$ on the message's key k and selects the least loaded worker among $h1(k)$ and $h2(k)$. To determine which one is the least loaded, the source only relies on internal information it has previously stored about the amount of messages the source has already sent to each of these two workers recently. Once load information is retrieved, the worker with the lowest load is adopted for processing the message. The authors argue that as long

as each source balances well its produced amount of load, no worker node can be overloaded even though a source cannot be aware of the amount of load sent to a node by the other sources. This technique offers good load balance by preventing a worker from being overloaded or underloaded compared to other workers, which can happen when using other partitioning schemes such as key grouping.

3.1.3 Cloud

Cloud computing separates the hardware side of a network from the software side by using virtualization. Hence, applications running in the Cloud can automatically scale depending on the amount of incoming requests or data to store. This process is done while remaining completely transparent to users. Thus, Cloud computing represents an efficient solution for Big Data storage, processing and analysis. One of the most popular services for Cloud computing is the Amazon Web Services (AWS) Cloud computing platform proposed by Amazon. Below, we present Auto Scaling, one of the load balancing solutions offered by AWS.

Amazon Web Services Auto Scaling

AWS [49] provides Cloud computing services featuring an auto-scalable infrastructure, if needed. Users can rent virtual machines (*instances*) providing computing and storage capacities. An Amazon CloudWatch sensor monitors instances to retrieve their load periodically (the granularity is defined by the user), by sending load state queries. Auto Scaling is a functionality responsible for adding or terminating some instances associated to an application running in the Cloud, if necessary. To do so, Auto Scaling relies on CloudWatch that is responsible for detecting if an application running in the Cloud is overloaded or underloaded according to the values (such as CPU utilization, network traffic and disk I/O usage) retrieved by its sensor. A threshold is associated to each metric and whenever a threshold is exceeded for a given time duration, CloudWatch triggers Auto Scaling to ensure the application uses the right amount of Amazon instances. Overall, the possible differences between the load balancing strategies used with Auto Scaling may concern three parameters. The frequency of the probes to retrieve load information may vary, as well as the metrics to monitor and the threshold associated

to each metric, used to determine an instance's load state.

3.2 Load Balancing in Structured P2P Systems for Data Management

Structured P2P systems are an efficient and scalable solution for data storage and retrieval in large distributed environments. However, many structured P2P systems for data management face the problem of load imbalance between nodes. These imbalances may be caused by different factors, and hence many different solutions have been proposed to address these issues.

In [50], Felber *et al.* propose to decompose into three main categories the existing load balancing solutions in structured P2P systems. The resulting taxonomy suggested by the authors is presented in Figure 3.1 (taken from [50]). The first category, *object placement*, refers to solutions for balancing the load caused by a bad distribution of skewed data or popular items. These strategies consist in assigning a given amount of keys/virtual servers to each peer so that they all store similar volumes of data and/or handle a similar number of queries. This generally implies either the replication of popular data items, the use of multiple hash functions to choose the best location for storing an item or replicate this item several times, or the reassignment of the identifiers managed by some peers. The second category, *routing*, includes strategies that modify the way messages are routed between peers in the overlay to prevent network traffic congestion. Typically, these approaches aim at creating/modifying/duplicating/shortening a particular path used to link two peers or to route a popular query. Finally, the third category, *underlay*, corresponds to building the overlay links according to the underlay topology, i.e. physically close machines in the underlay become neighboring peers in the overlay, to minimize the network traffic. We will not go into further details on this third category as it is outside the scope of the issues we want to address, namely the poor distribution of skewed/popular data that may overload some peers.

Throughout this section, we review some various and relevant load balancing

strategies for structured P2P systems.

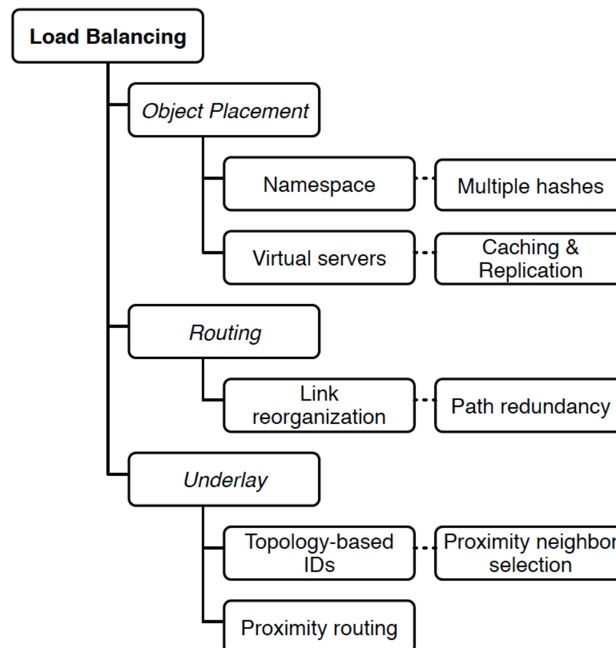


Figure 3.1 – Taxonomy of load balancing solutions in P2P systems (taken from [50]).

3.2.1 Load balancing for RDF data

To our knowledge, there exist very few load balancing strategies set in the context of RDF data storage. In the following, we present the two most well-known solutions from the literature, that both aim at addressing the overload caused by the bad dissemination of highly frequent RDF terms.

RDFPeers

In RDFPeers [41], two solutions are proposed to deal with load imbalances when storing RDF triples in an order-preserving manner at three different locations: $successor(hash(s))$, $successor(hash(p))$ and $successor(hash(o))$.
With such architecture, a biased data distribution may appear if some values of s, p or o are more frequent than others. We argued in Chapter 2 that predicates

usually have much less distinct values. The authors of RDFPeers relate that 16% of the triples in one of the datasets[1] used for their experiments contain the same predicate *rdf:type*, used to represent the type of a resource. This means that all these triples would be stored by the same peer when indexing them with their predicate value. Consequently, to deal with the high popularity of some predicate values, the authors propose to stop indexing a triple on its predicate if its value becomes too frequent. Each node counts how many times it stores the same predicate and if this counter exceeds a specified threshold, the node refuses to store new triples indexed on this predicate value. If the node later receives a query for this refused predicate, a refusal message is returned to the sender that will have to solve its query using the other parts of the triple. This strategy, albeit radical, offers the advantage of not overloading a peer with identical values to store and also limits the load due to query processing on irrelevant and highly frequent terms.

This first solution may not address all load imbalances in terms of triples stored per peer as triples are stored in an order-preserving manner, thus they cannot be uniformly distributed among nodes. To improve data dissemination, the authors propose a *successor probing* scheme. This scheme consists in adding a peer at the most loaded location among d sampled locations in the overlay. A peer that wants to join the network computes d possible locations in the ring where it could join and contacts the d existing peers at these locations in order to estimate how many items the new peer would retrieve when becoming the predecessor of one of these peers. Finally, the new peer joins as the predecessor of the most loaded peer contacted in order to decrease its load.

Battré *et al.*

Battré *et al.* [43] propose a different approach to address the bad dissemination issue for popular RDF terms like *rdf:type*. Their strategy consists in creating overlay trees onto the existing overlay. A peer can trigger load balancing when its load exceeds a threshold. To balance its load, an overloaded peer p_o splits its current dataset in half. One part remains stored by p_o and the other half is sent to p_u, an

[1] `http://www.dmoz.org/about.html`

underloaded peer previously probed. In practice, these triples should not be stored by p_u because their hashed value in the identifier space refers to another peer (p_o). Thus, p_o has to maintain a reference to p_u in order to redirect queries for these triples to p_u. In order to keep p_u consistent, and as peers can maintain several databases in the proposed approach, p_u stores p_o's triples in a *remote* database, containing triples that p_u should not store according to the DHT. Therefore, this strategy requires more hops to solve queries, as the peer matching the DHT rules may have sent part of its data to another peer, and this peer may have also sent part of this dataset to another peer, and so on and so forth. Nonetheless, this is an interesting strategy because it allows to disseminate identical terms like *rdf:type* across many peers, which is normally impossible when indexing *rdf:type* because *hash*(*rdf:type*) always refers to the same coordinate in the identifier space. However, this implies that the peer responsible for *hash*(*rdf:type*) is the only one to know where the rest of the triples featuring *rdf:type* are located, which may be problematic in case this node fails. Also, this peer will necessarily continue to receive all queries for *rdf:type*, which generates processing load for this peer, and more communications are needed to retrieve all the desired triples, which increases network traffic.

In the following, we expand our inverstigation to other load balancing strategies used by distributed systems for data storage, not especially dedicated to RDF.

3.2.2 Virtual servers reassignment

Rao *et al.*

Rao *et al.* [51] suggest three different strategies based on virtual peers to address the issue of load imbalance in P2P systems that provide a DHT abstraction. Unlike traditional P2P networks where one peer is deployed per node, virtual peers are an abstraction allowing several peers to be hosted on a same physical node. Upon the detection of an underloaded or overloaded peer, virtual peers are reassigned to other nodes in order to maintain the machine load under a given threshold. This paper proposes a general solution, not especially dedicated to data load balancing. The aim is to address any kind of load imbalance issue, whether it concerns

storage, CPU or bandwidth, that may cause a bottleneck in the system.
Each physical node is responsible for one or more virtual servers, whose load is bounded by a predefined threshold. A node is considered as imbalanced depending on this threshold: heavy if its load is above, light otherwise. The proposed solutions are meant to transfer the load between heavy and light nodes by moving virtual peers only. The first scheme, called *one-to-one*, involves two peers to decide whether a load transfer should be performed or not. A peer simply contacts a randomly chosen peer, and both exchange their load information. If one of them is heavy and the other one is light, then a virtual server transfer is initiated. The second scheme (*one-to-many*) relies on directories indexed on top of the existing overlay. Some nodes store piggybacked load information from light nodes on these directories. When a node holding a directory receives a message from a heavily loaded node, it looks at the light nodes in its directory to transfer the heaviest virtual server from the heavily loaded node to a lightly loaded one. Finally, the third variant (*many-to-many*) matches many heavily loaded nodes to many lightly loaded nodes, still using directories. A node holding a directory receives load information from both heavy and light nodes. This node periodically performs an algorithm to calculate how to balance the load between all these nodes. Solutions specifying which virtual servers should be transferred to which nodes are then sent to the concerned nodes.
The load balancing strategies for virtual peers presented in this paper are simple and provide satisfying results, especially for the schemes using directories. For instance, the third strategy made 95% of the heavy nodes light after one probe to a node holding a directory. On the negative side, the experiments are performed in a static system, which means there is no data insertion/update/deletion during experiments, as well as no peer joining or leaving the system. Furthermore, the one-to-one scheme does not perform well concerning the time it takes to achieve balance, as peers are randomly sampled, and requires up to 20000 probes (in a network made of 4096 nodes) before all heavy peers become light, with 75% of these probes being unsuccessful in finding an underloaded peer.

Godfrey *et al.*

In [52], Godfrey *et al.* present their work consisting in a dynamic version of the solution proposed by Rao *et al.*. Their goal is to provide an efficient load dissemination among nodes even in case of churn and frequent insertion/removal of items, including skewed data. The proposed load balancing operations are performed either periodically or under emergency. A periodic rebalancing is performed similarly to the *many-to-many* scheme proposed by Rao *et al.*: a peer holding a directory periodically computes an algorithm to decide which virtual server should be transferred to which node, based on load state information periodically sent by nodes. Conversely, the *emergency* scheme, which is the main contribution of the paper, is immediately triggered by a peer whenever its load (a combination of metrics including disk space, processing and bandwidth consumption) exceeds an internal threshold. Such situation may typically occur if a large dataset of skewed values is inserted and has to be indexed by the same peer. Under these circumstances, the overloaded peer contacts a directory that immediately computes a solution specifying which underloaded nodes could receive some of the overloaded peer's virtual servers. Therefore, this is a more reactive approach than what is proposed by Rao *et al.* with their load balancing solutions computed periodically only, regardless of the possible events that might happen in the overlay (e.g., node arrival/departure or insertion of large datasets).

3.2.3 Data/Node replication

HotRoD

In [53], Pitoura *et al.* present HotRoD, a locality-preserving DHT-based architecture for data storage. This system, based on a Chord-like ring, stores data items in the form of tuples made of k attributes. Tuples are ordered in the overlay according to the value of at least one of their attributes (k, at most). This value, when hashed using an order-preserving hash function, becomes the key identifier of the tuple. As in a Chord overlay, a tuple t is located at $successor(t_{key})$.

With such architecture, load imbalances may appear regarding data placement

and some peers may become overloaded due to the popularity of some of their items. Thus, to deal with popular queries execution, the authors introduce a replication scheme. Their goal is to replicate popular items or ranges of popular items while maintaining the natural ordering of data in order to still execute range queries in a minimum number of hops. However, if an overloaded peer places replicas in its neighborhood to preserve the order of values, this may not be efficient if its neighbors are also overloaded. Hence, the proposed replication scheme consists in creating virtual rings for storing replicas of popular items over the regular DHT ring.

Load balancing is triggered when a peer is *"hot"*, i.e. when a peer estimates (this is done periodically) it is overloaded because it has processed too many queries in a given time interval. As the paper focuses on range queries execution, having a *"hot"* peer in a defined arc of peers in the ring (including some of the *"hot"* peer's successors and/or predecessors) means this arc is *"hot"*, too. Consequently, peers in this arc create replicas of some of their items at underloaded peers, in a virtual ring that can be accessed from the regular ring. Arc of peers are replicated instead of a single peer in order to avoid jumping from one ring to another too often when processing popular range queries. In order to limit replication costs, not all items of these peers are duplicated but only the most popular ones having few or no replica yet. Concretely, this means that an underloaded peer, responsible for storing a given range of values in the regular ring, may be responsible for a different range of values (the popular ones) in the virtual ring in order to store replicas. This architecture is depicted in Figure 3.2 (taken from [53]). This figure considers a system made of two rings: Ring 1 is the regular ring and Ring 2 is the virtual one. Popular data items, ranging between values 604 and 1910, are replicated in Ring 2 on underloaded nodes from Ring 1. For instance, node 14720 is underloaded in Ring 1 because it is responsible for non-popular values comprised in $[2862; 3916[$. However, it manages a more popular interval in Ring 2: $[814; 1632[$. When executing a range query on popular items, a peer in Ring 1 may jump to Ring 2 in order to reach node 14720 instead of node 4912, when looking for values comprised in $[814; 1632[$. For example, in Figure 3.2, node 11448 is looking for a set of values ranging from 1000 to 2000. Solving this query on Ring 1 would mean that the query has to be forwarded to node 4912 which requires

several hops whereas the query initiator has a neighbor (node 14720) that matches the requested interval on Ring 2.

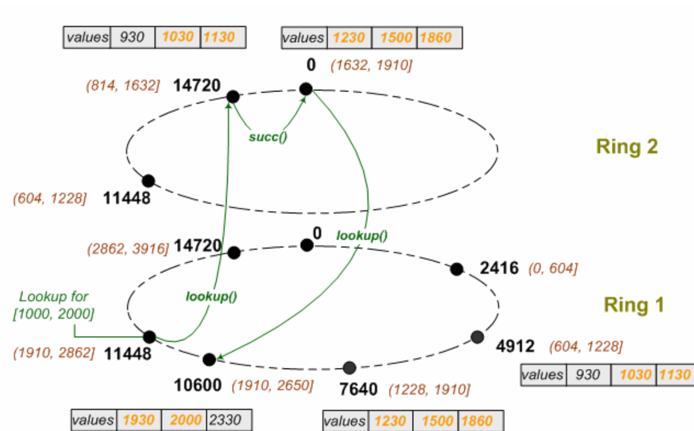


Figure 3.2 – HotRoD architecture (taken from [53]).

Although the use of replication increases fault-tolerance and may prevent some peers from being overloaded by queries, this load balancing strategy suffers from the usual drawbacks associated with replication schemes. Such an approach leads to more disk space consumption and has high consistency constraints on updates. Moreover, the use of virtual rings adds one more level of complexity to this strategy, concerning updates on items and the way these new rings are added, especially in the case of several overloaded peers concurrently asking for the creation of new rings.

Meghdoot

In [54], Gupta *et al.* present Meghdoot, their publish/subscribe system. Data is in the form of tuples of n attributes encoded as character strings, integers or floating-point numbers. Users can specify subscriptions over one or more attributes that match some constraints. Events are a set of values sent to the system, each of them being associated to one of the n attributes managed by the system. Meghdoot is based on a CAN overlay made of $2n$ dimensions, where dimensions $2i$ and $2i-1$ are dedicated to the possible values of attribute i. Thus, an event or a subscription is

a point in the $2n$-dimensional space, managed by the peer whose zone encompasses this point.

As a peer storing subscriptions is also responsible for providing all the matching events for these subscriptions to their subscribers, some peers may become overloaded if they manage too many subscriptions. Moreover, if similar events (same attributes with similar values) are regularly sent to the system, similar paths will be used when propagating these events to the concerned CAN zone. Consequently, peers located on a popular propagation path may be too loaded because they have too many events to route. Therefore, peers in Meghdoot can experience two different kinds of overload.

To address these issues, the authors exploit both the characteristics of CAN and their publish/subscribe system properties to balance the load when new peers join the system. Each peer periodically propagates information about its load state to its neighbors. When a new peer wants to join the system, it contacts a known peer in the system, responsible for locating the heaviest loaded peer. The authors distinguish subscriptions load from events load. To balance subscriptions matching load, the idea is to split a heavy peer's zone so that its number of subscriptions is evenly divided with the peer that joins. This zone splitting strategy is similar to the default load balancing scheme of CAN consisting in splitting the zone of an overloaded peer into two zones of equal surfaces, managed by two peers. However, the difference with Meghdoot is that the two new zones may not have even surfaces as the goal is that they both contain the same number of subscriptions, which is much more interesting regarding load balancing.

The second solution, to address event propagation load imbalance, creates alternate propagation paths by using replication: when a new peer p_j joins a peer p_i overloaded by events, the zone from p_i is replicated on p_j (including its subscriptions). In addition, the neighbors are updated to keep track of p_j in a replica list. Finally, events are balanced during the propagation of an event to be matched with candidate subscriptions by picking, on the peer that executes the routing decision, one of the many possible paths (i.e. one of the replicated peers) by following the round-robin principle. This replication strategy improves load balancing, data availability and performance.

One of the main benefits of the solutions proposed by Meghdoot is that routing

the load that has to be moved from an overloaded peer to a new peer only requires one hop as they are neighbors. However, the arrival of the new peer modifies the network topology, which has a cost. Also, with these solutions, there is no way to offload an overloaded peer as long as no new peer joins the system.

3.2.4 Keys reassignment

Reassigning a set of keys to a peer may be used to create replicas of popular items, as we have seen with the approach proposed by Pitoura *et al.* and their system HotRoD. However, this technique can also be used for skewed data dissemination, as discussed below. We present two different techniques based on key reassignment, the first one implying to modify the network topology (*leave-join* mechanism) while it is not mandatory in the second one.

Mercury

In [55], Bharambe *et al.* present Mercury, a system made to support range queries on top of a structured P2P network constructed by using multiple interconnected virtual ring layers where each ring is named a hub. In Mercury, data items are in the form of tuples, made of several attributes with their associated schema. Each hub manages the indexation of an attribute from this predefined schema. Thus, a hub can be seen as one dimension in a d-dimensional virtual space, where d is the number of attributes defined in the data schema. Consequently, a physical node can be a member of several hubs.
Mercury does not use hash functions for indexing data and suffers from non-uniform data partitioning among peers as data requires to be assigned continuously for supporting range queries. Owing to this bad data distribution, the authors propose load balancing mechanisms based on low overhead random sampling to create an estimate of the data value and load distribution. Basically, each peer periodically sends a probing request to another peer using random routing. This offers a global system load assessment whose values are collected into histograms maintained on peers. Using this information, a heavily loaded node can contact a lightly loaded node and request it to leave its location in the routing ring (and

hence hand over its data to a neighbor) and re-join near the location of the heavy node (*leave-join* mechanism), to become its predecessor. Finally, the underloaded peer captures half of the keys the heavy peer (its successor) is responsible for.
The authors show this approach is enough for effective load balancing because their system topology is an expander graph with a good expansion rate. In other words, with a small number of edges in their network topology, everyone can reach other edges by many paths. However, due to the architecture of Mercury, the leave-join mechanism implies many links to be repaired, especially when a node leaves its location. In a classical ring overlay, successors/predecessors links have to be changed when a node leaves. With Mercury, it is also necessary to update inter-hub links and long-distance links (equivalent to finger table links in Chord). Therefore, it may take some time before all links in the overlay are consistent in case of many peers leaving their location in a short time interval, which may affect query lookups.

Konstantinou *et al.*

In [56], Konstantinou *et al.* present their load balancing solution designed for P2P range-queriable systems. Their mechanism works on top of a skip graph [57] where peers are placed in an order-preserving way regarding the keys they manage. This placement preserves the natural ordering of data in order to perform efficient range query processing. However, as the use of an order-preserving storage technique cannot evenly distribute data among peers (unlike when using a randomizing hash function), some peers may become overloaded. To address this issue and maintain good performance of the system, usually either data or node migration is used. Both strategies have their drawbacks: data migration might take time to achieve balance and requires to move large amounts of data, while node migration is also costly as it modifies the network topology. The solution proposed in this paper combines both strategies. The authors insist on the fact that combining both node and data migration would be less costly in terms of data transfer and information exchange than when using only one of the two schemes. Each peer has its own internal load threshold (depending on its capacities) and whenever this threshold is exceeded, a peer can transfer some of the keys it manages to its direct neighbor, this is called the *NIX* approach in this paper, until it is not overloaded anymore.

However, this may not always suffice if several neighboring peers are all overloaded. In such a case, the overloaded peer sends probing messages throughout the network until it finds an underloaded peer that can migrate close to it and capture some of its keys: this is the *MIG* approach. Hence *MIG* also requires data migration in the end, like *NIX*. For both approaches, the number of keys to be acquired by the neighbor (*NIX*) or the underloaded peer (*MIG*) is calculated in such a way that the number of items to be transferred between the two peers is minimized.

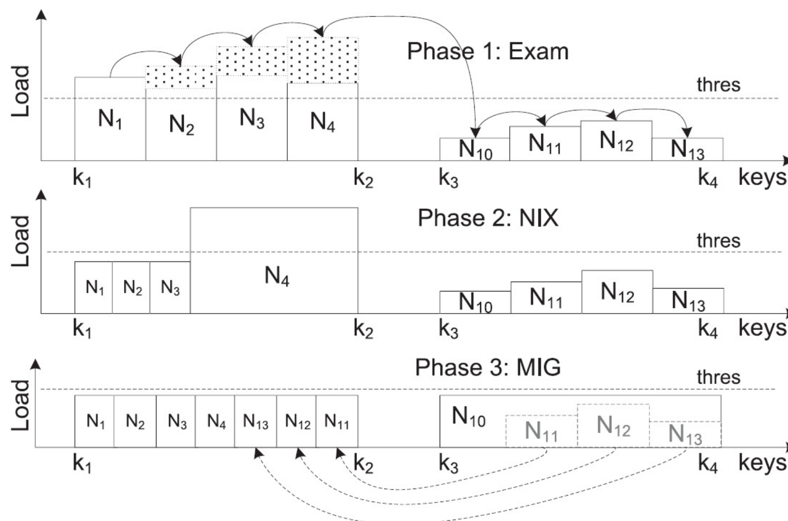


Figure 3.3 – *NIXMIG* approach proposed by Konstantinou *et al.* (taken from [56]).

Figure 3.3, taken from [56], describes the combined *NIX* and *MIG* approaches proposed in this paper. Node N_1 is overloaded and sends part of its data to its neighbor N_2, which becomes overloaded. The same happens when part of N_2's data is moved to N_3 and then from N_3 to N_4. As this neighborhood is well overloaded and no local solution seems to be found, N_4 contacts an underloaded node, N_{10}, which reserves nodes from N_{11} to N_{13}, so that they leave their position and re-join as N_4's successors to capture part of its keys. As a result, N_4's key interval is reduced by 75% at Phase 3, while N_{10} takes charge of the range of values previously managed by nodes N_{11} to N_{13}.

In summary, this is a simple but efficient solution to balance the load in a "wave-like" fashion when storing data in an order-preserving manner. Two common load

balancing strategies, consisting in either moving data to a neighbor or asking an underloaded peer to become the neighbor of an overloaded one, are combined in this paper. However, a peer can only balance its load with one neighbor at a time, hence it may take a long time before a full rebalancing of the network is achieved. Moreover, as an overloaded peer mandatorily needs an underloaded neighbor to move its data, this may imply to change the network topology very often. Finally, this approach is particularly well-suited for skip graphs, whose topology is quite simple, but may not be compatible with other P2P overlays based on more complicated architectures, like Kademlia (binary tree using the XOR metric) or CAN (n-dimensional torus).

Chapter 4

Variable Hash Functions

Contents

4.1 State of the Art 50

4.1.1 Multiple hash functions for popular items 50

4.1.2 Multiple hash functions for data dissemination 53

4.2 Principles and Interest of our Approach 55

4.2.1 CAN for data storage 57

4.2.2 Skewed data, skewed distribution 59

4.2.3 Variable hash functions for overloaded zones 60

4.3 Hash Function Update Process 64

4.3.1 Update propagation 64

4.3.2 Bound reduction in a torus-like topology 67

4.3.3 Optimal update propagation in a torus-like CAN 68

4.3.4 Concurrency and message filtering 71

4.3.5 Data movement and lookup 73

4.3.6 Main rules 77

4.4 Summary 79

Hashing is commonly used by famous P2P systems like Chord [18], CAN [19], Pastry [21] and Kademlia [20]. This technique is also used by many distributed

systems for data storage, like Cassandra [33], Oracle NoSQL Database [58] and HBase [34]. In DHT networks, hash functions are responsible for data placement. Many P2P systems use a DHT to locate where a data item should be stored in the network. Each node only has a partial knowledge of other peers located in the overlay. When a peer wants to insert or retrieve an item, it applies a hash function on this item and looks at its DHT in order to find coordinates corresponding to where in the overlay the item should be stored. Consequently, the hash function being used is directly responsible for both determining the good distribution of data across nodes and also the efficient retrieval of resources in the overlay.

The use of a single hash function is usually a natural solution to obtain a well-balanced network regarding data distribution, especially when the hash function randomly disseminates data over peers. However, under certain circumstances that we detail in this chapter, like the use of an order-preserving hash function and/or skewed key values to index, using a single hash function may lead to load imbalances among peers. This is why the use of multiple hash functions is sometimes suggested in the literature, principally as a solution to address two issues causing load imbalance, as described below in Section 4.1. Applying several hash functions is usually proposed as a replication scheme for popular items, and more rarely as a way to distribute data items among peers.

In this chapter, we first present existing load balancing strategies based on multiple hash functions in structured P2P networks. Then, we introduce our own approach using variable order-preserving hash functions to dynamically distribute skewed datasets in structured overlays. Finally, we detail our solution along with the rules to follow when implementing it in order to maintain a consistent overlay.

4.1 State of the Art

4.1.1 Multiple hash functions for popular items

Multiple hash functions are used in exising DHT networks as a solution to address two load imbalance issues. The first issue concerns the popularity of items. A

data item may be more popular than others and, consequently, the peer storing this item has to face an important number of queries, which affects its processing capacity. To speed up response time when a request for a popular item is made, some load balancing strategies replicate a popular item by applying n different hash functions on this item, in order to replicate it at n different locations. For example, the original CAN paper [19] suggests this approach to improve data availability and also to reduce the path length when routing a query. This way, when a peer is looking for an object obj, it can apply n hash functions on obj. Each hash function maps obj to a different CAN coordinate and a peer looking for obj can choose the hash function associated with the nearest coordinate to its location in order to minimize the number of hops to retrieve obj. However, this technique requires that all peers know all the different hash functions used to place data in order to be able to choose from where to retrieve it afterwards.

Other papers propose similar replica placement strategies based on multiple hash functions to deal with popular items. In [59], Xia *et al.* consider a P2P system for file storage where a peer can replicate a popular file at a location found by applying on the file a uniform hash function currently unused in the overlay. The paper presents how to choose this new hash function along with mechanisms allowing the other peers in the overlay to find existing hash functions they can apply on the file they are looking for. The paper presents a system where at most m uniform hash functions may be used to replicate a popular file. Let us consider a popular file currently replicated at i different locations, using $h_1, ..., h_i$ hash functions. Every time a file f exceeds a popularity threshold on a peer p, p has to replicate this file at another location using h_{i+1}. As f's i replications may have been triggered by different peers, p may not know the value of i and hence what is h_{i+1} that it should apply on f. To find f's current number of replicas, the paper proposes an algorithm that works as follows: p applies x hash functions $h_1, ..., h_x$ on f until h_x points to a peer that does not store f, i.e. $x = i + 1$. Then, p can create the $i + 1^{th}$ replica of f at the peer matching $h_{i+1}(f)$. To retrieve a file f, peers use a random binary search algorithm: the peer applies $h_r(f)$ where r is a random value comprised between 1 and m. If f is not found when using this hash function, the peer applies $h_{r2}(f)$ where $r2$ is comprised between 1 and

r, and so on until the peer finds f. Concretely, this algorithm consists in reducing the number of possible locations for a resource after each iteration, until narrowing down to the sequence of hash functions that were used to index the item.

In [60], Mu *et al.* present their approach to dynamically increase or decrease the number of replicas for a popular object. The authors assume a peer can evaluate which of its items are the most popular, in terms of load consumption for a set of predefined load criteria. Every time a node reaches a given load threshold, it creates a ith replica of its most popular item by using hash function h_i and sends the replica to the corresponding peer. Each original object embeds a counter number (equal to i) indicating the number of replicas existing for this object. The authors also assume a peer can distinguish its own original items from those that are replicas received from other peers. Whenever the load of a peer caused by one of its own popular objects decreases under a given threshold (i.e. the object becomes less popular), the peer can launch a removal process of one of its replicas. Concerning the retrieval of items, a peer looking for an object *obj* for the first time can only use the default hash function known by all peers in the overlay. When retrieving the original item, the peer also retrieves its associated counter number and thus knows the range of hash functions used to replicate this item. If, later, the same peer wants to retrieve *obj* again, it will be able to contact one of the i locations of *obj* by computing a hash function between h_1 and h_i. The main drawback of such strategy is that peers can only benefit from the use of replicas if they need to access the same popular item several times (assuming they do not make a copy of it because its value varies) and the popularity of this item has not decreased in the meantime (in this case some replicas might have been deleted).

In [61], Wu *et al.* use the concept of multiple hashes for data storage and retrieval in the Kademlia [20] P2P network. The authors consider a system where objects are indexed in the overlay according to a keyword value. If all peers use the same hash function to index objects, a bad dissemination of objects occurs when most of them are associated with the same subset of popular keywords. Indeed, a popular keyword *word*, associated with n different objects, generates n identical keys for all these objects if the same hash function is always applied on *word*,

which may overload the peer responsible for the corresponding key. To address this issue, the authors propose that the same keyword may be hashed different times with the same hash function, in order to produce different keys. Each of these keys refers to the same keyword but points to a different peer and hence objects associated with the same keyword can be indexed on different peers instead of only one. For example, $hash(word)$ is the default hash function of the system for keyword $word$ and derived hash functions h_2 and h_3 would respectively correspond to $hash(hash(word))$ and $hash(hash(hash(word)))$. Up to N hash functions can be used, thus the same keyword may be stored by N peers (assuming each different hash function maps a different peer). The disadvantage of this method is that when a peer is looking for resources associated with a given keyword, there may be results in up to N peers in the system due to the fact that the desired keyword might have been hashed up to N times. Thus, a query must be sent to N peers and the simulation results presented in the paper show that as N increases, the traffic overhead becomes more and more important. This is why the authors estimate 7 different hash functions are enough to balance the load without causing too much overhead.

4.1.2 Multiple hash functions for data dissemination

Byers *et al.* suggest in [62] the use of multiple hashing functions for a different purpose than popular items replication. Their approach, presented on the Chord [18] P2P network, aims at addressing load imbalances in terms of items stored per peer, by trying different hash functions on an item to insert it at the least loaded location. To do so, the paper presents a variant of the power of two choices paradigm [48]. Basically, the power of two choices paradigm consists in applying two hash functions picked at random on an item's key to eventually pick the least loaded node out of the two possible locations to store this item. This paper extends this paradigm to the use of two or more hash functions to compute the potential item locations.

A node that wishes to insert an item applies d hash functions picked at random on the item's key and gets back d identifiers (each hash function is assumed to map items onto a ring identifier). Afterwards, a probing request is sent for each

identifier computed previously and the peers managing the identifiers answer with their load. Once load information is retrieved, the peer with the lowest load p_{low} is adopted for indexing the item. The other $d-1$ peers that were contacted but not selected receive a redirection pointer (key space identifier) to p_{low} for the corresponding item.

When a peer wants to retrieve an object obj but does not know which hash function has been used to index obj, the peer applies on obj's key a random hash function $h_r(obj_{key})$ among the d possible hash functions that were applied previously to store obj. Even if h_r does not point to the peer storing obj, the peer receiving this query necessarily owns a redirection pointer to obj. Thus, a lookup can be achieved by using only one hash function among d at random and it may take only one more hop to reach the desired resource. However, this technique implies that all peers are aware of the d possible hash functions applied when indexing resources, in order to use one of them, albeit randomly chosen, when looking for an item.

Following the same principle, load-stealing and load-shedding strategies can be used, too. An underloaded peer should be able to take in its possession items for which it owns a redirection pointer, whereas an overloaded peer can hand over an item to a lighter peer using a redirection pointer. The experimental results show that using two hash functions ($d = 2$) is enough to achieve a better load balancing with their two choices strategy rather than using a limited number of virtual peers.

In summary, this is an effective solution to prevent load imbalances. Unlike many load balancing strategies, this one does not exclusively focus on reducing the load of overloaded peers. Instead, it prevents this situation from happening by evenly distributing the load on peers from the beginning. It is also efficient to distribute skewed datasets, although the use of a single randomizing hash function is generally sufficient. However, the natural ordering of data, which is important to simplify range queries processing, cannot be preserved. Moreover, several probes are necessary every time an item has to be inserted, which increases the number of communications and may slow down the system when inserting a large number of items.

4.2 Principles and Interest of our Approach

The aforementioned solutions based on multiple hashes all use multiple randomizing hash functions. Random hashing is commonly used because it is generally an efficient technique to uniformly distribute the load among peers. However, the use of a random hash function for placing data implies a random distribution of items, which also has drawbacks. Indeed, the execution of range queries on non-contiguous peers necessarily increases query execution time to retrieve all matching results, possibly distributed all over the network. This is why some distributed systems for data storage and retrieval tend to use an order-preserving hash function to distribute data across nodes. These systems can be RDF repositories like the EventCloud, RDFPeers [41] and GridVine [63] or, more generally, P2P-based solutions for data storage like [64] [65] [42]. Data is no longer randomly distributed, and syntactically close items can be retrieved in a minimum number of hops. Peers still use a hash function to determine an item's location: the function is not random but uses the natural value of the item (e.g., its lexicographic value) to generate its coordinates in the overlay's identifier space. Unfortunately, this technique cannot always guarantee a satisfying distribution of data among peers since such a hash function cannot efficiently disseminate lexicographically skewed data, like RDF terms [10], made of the same or close Unicode characters.

It is also interesting to note that, in most systems, all peers use the same hash function. Thus, even though peers do not have an overall knowledge of the system, they are all able to estimate where in the network an item should belong to. Therefore, the use of the same hash function by all peers can be seen as a centralization of information about items location, where peers act as if they were asking an oracle to provide them an item's coordinates. However, as the use of an oracle is an obstacle to scaling, we believe it should be useful to take advantage of some of the benefits offered by P2P systems, like decentralization, to build a *decentralized* hash function as well. More concretely, we believe all peers should not necessarily use the same hash function in the overlay to determine an item's coordinates.

When using multiple hashes, we have shown that existing strategies require that, when looking for a particular data item, a peer either statically picks one (Byers *et al.*) or more (Wu *et al.*) hash functions among d known by all peers, or dynamically learns about the hash functions used by other peers to index the item (Xia *et al.* and Mu *et al.*). Unlike these existing strategies, we propose a solution based on variable hash functions where each peer can use its own hash function to place and retrieve data. Moreover, it is not required that all peers learn about all the hash functions that can be used in the overlay. Using our approach, as peers do not share the same hash function, routing is done on the go, from one peer to another, and each peer applies its own hash function when receiving a message, to determinate its next recipient. Therefore, the initial sender of the message potentially has no precise idea about where in the system the message will be processed. Like Byers *et al.*, we propose the use of multiple hash functions to improve data distribution and avoid peers from being overloaded. Unlike Byers *et al.* and the other strategies based on multiple hashes presented above in Section 4.1, we modify the interval of keys a peer is responsible for, i.e. we perform keys reassignment on overloaded peers. However, to do so, we do not modify the network topology. Instead, we modify the hash function used to map an interval in the overlay's identifier space so that less hashed keys map into this interval. Moreover, we aim at addressing the skewed data dissemination issue in systems relying on non-randomizing hashing to place data, by using variable *order-preserving* hash functions. We allow overloaded peers to independently modify their own hash function for placing data, hence it would be possible to provide a much more satisfying data distribution when hashing skewed datasets according to the lexicographic order, while preserving this natural ordering of data along with all its benefits. To our knowledge, this is the first solution using multiple order-preserving hash functions in a structured overlay.

In this section, we present the benefits of our strategy to balance the load among peers and how it can be applied on the CAN overlay. However, it is absolutely conceivable to use this technique on other P2P systems based on a DHT, as we will show later in this thesis.

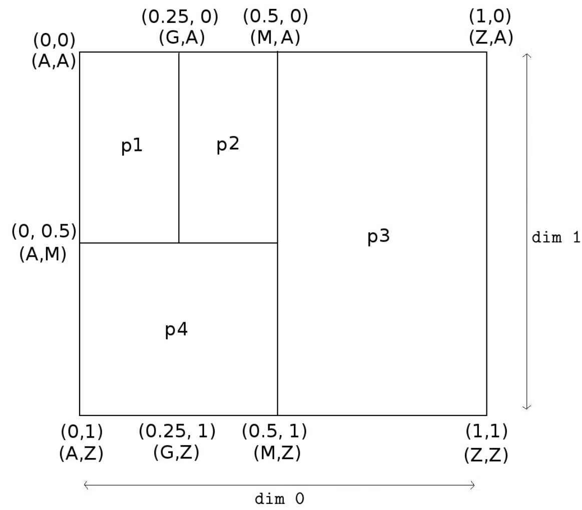
4.2.1 CAN for data storage

In this chapter, we will consider a 2-dimensional CAN only, and to show the interest of our load balancing strategy, present how this system can improve the distribution of data made of Unicode characters according to the lexicographic order. We will consider data items made of 2 Unicode-encoded attributes in the form of $< attribute^0; attribute^1 >$, each value of an attribute being associated to one of the CAN dimensions (0 or 1). For all dimensions, minimum and maximum Unicode values U_{min} and U_{max} are set to determine the Unicode range that can be managed within the CAN (in Figure 4.1(a), U_{min} is equal to A and U_{max} is equal to Z)[1]. A Unicode value is associated to each CAN bound of a peer and a peer is responsible for storing Unicode values falling between these bounds on each dimension. For example, in Figure 4.1(a), $p1$ is responsible for data between $[A; G[$ (corresponding to CAN-based interval $[0; 0.25[$) on the horizontal dimension, and $[A; M[$ (CAN interval $[0; 0.5[$) on the vertical one.

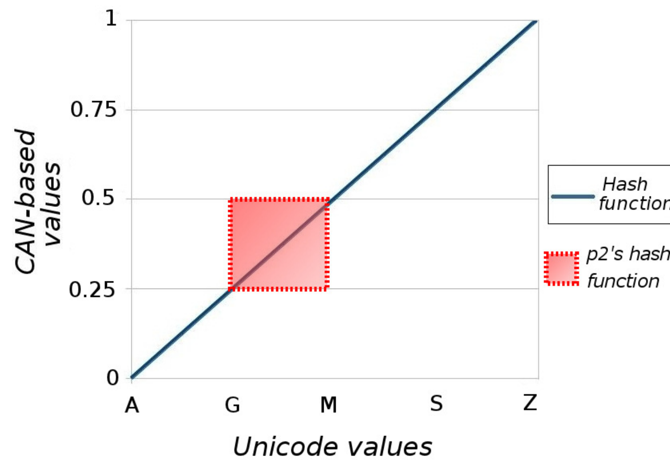
When a peer receives a new data item to insert or a query to execute, it has to convert the Unicode-encoded values into numerical values to check whether it is responsible for this item/query or not. For example, the string *label* in the context of a CAN storing worldwide data (wide Unicode range, up to Unicode character number 2^{20}) corresponds to Unicode value 108.00009632119509 (i.e. at the far-left of the CAN). Basically, the function we use to obtain such value iteratively converts each Unicode character into a codepoint value and divides these codepoints by U_{max}. By default, strings made of Latin characters have a low value, whereas strings made of any East Asian characters for example have high values (close to U_{max} and hence by default close to C_{max} as well) because such characters are located towards the end of the Unicode table of characters.

To each CAN-based interval $[min_p^d; max_p^d[$ of a peer p on a dimension d, corresponds a Unicode interval $[Umin_p^d; Umax_p^d[$. As a consequence, the mapping

[1] For more clarity, we use a single character to represent each bound's value. However, the real value is a numerical transformation (using the codepoint values) of a string of characters, for example Z actually corresponds to $ZZZZ$... which gives a codepoint value approximately equal to 90.



(a) Example of a 2-dimensional Unicode CAN storing items in the form of < $attribute^0; attribute^1$ >. Dimension 0 is dedicated to $attribute^0$ values and dimension 1 to $attribute^1$ values.



(b) Standard hash function on dimension 0.

Figure 4.1 – Standard hash function of a CAN, that determines Unicode values between $[A; Z]$ to be associated with the CAN bounds of peers on the interval $[0; 1]$.

relation between p's CAN interval and p's Unicode interval can be seen as the hash function applied by p for dimension d. By default, the function linearly matches the minimum Unicode value that can be stored within the CAN U_{min} (resp. U_{max}) to the minimum CAN-based value C_{min} (resp. C_{max}). The standard hash function that can be associated with the CAN of Figure 4.1(a) for dimension 0 is shown in Figure 4.1(b). This CAN manages Unicode values from A to Z within its interval $[0; 1]$. Representing this mapping on a graph means that A has to be associated with 0 and Z with 1. The function describes which CAN coordinate is associated with which Unicode value, hence each peer is associated with a segment of the function ($p2$'s segment is highlighted in Figure 4.1(b)). For instance, "G" corresponds to CAN coordinate 0.25 on the hash function graph, which means G is a value managed by peer $p2$ on dimension 0 because its CAN interval is $[0.25; 0.5[$. Thus, the hash function provides CAN coordinates that help determine where a Unicode-encoded data item should be stored in the overlay[2].

4.2.2 Skewed data, skewed distribution

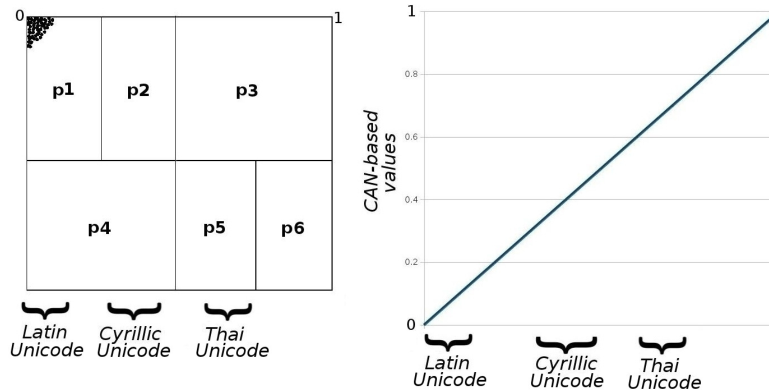


Figure 4.2 – Default hash function inefficient to disseminate data items (represented as black dots at the top-left corner of the CAN).

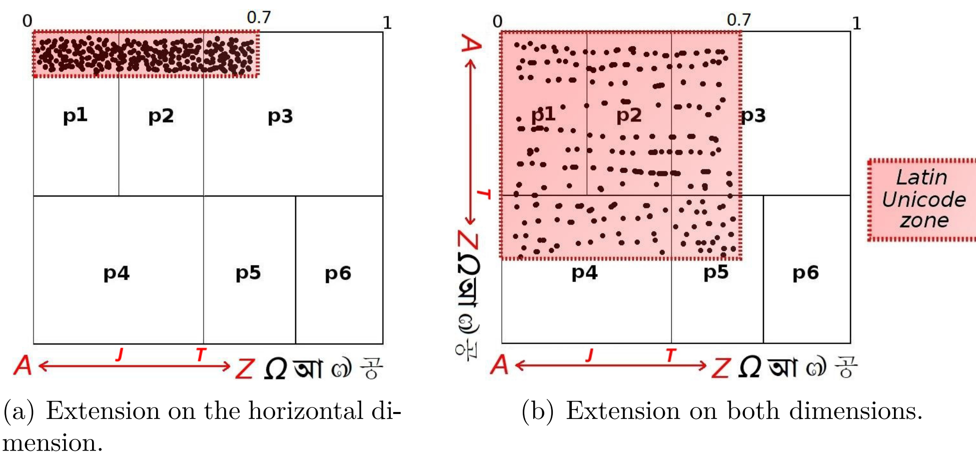
[2]In the implementation, a peer does not need to perform a computation based on a segment of this straight line to evaluate whether an item belongs to its zone or not. The peer can directly compare the item's codepoint value with the Unicode values of its bounds. However, this is a equivalent alternative and we use this graphical representation to concretely show the impact of our solution in the overlay.

The order-preserving storage technique presented above suffers from a major drawback regarding data distribution. Indeed, having a system covering the whole Unicode range means potential overloaded areas may appear, depending on data distribution. Figure 4.2 shows a system where only items made of Latin characters are currently stored, which means only a very small area of the CAN is always targeted when storing or querying data. The most basic (no accent) Latin Unicode range only includes about 100 different characters whereas there are about one million characters in the whole Unicode table, which makes Latin-encoded values very skewed. In consequence, peer $p1$, whose zone includes the small Latin interval, becomes overloaded, while the rest of the network stores nothing as it is dedicated to other Unicode characters (Cyrillic, Thai, etc.). A solution could be to dedicate the entire network to Latin Unicode data storage. However, as an efficient system should be able to deal with any dataset from anywhere in the world, this could be problematic in case of a burst of incoming large data sets in a different alphabet. Moreover, there are also many symbols like mathematical operators and emoticons, having their own range in the Unicode table of characters, that should necessarily be admitted to the system.
Based on this observation, our goal is to allocate more space for skewed values in the CAN, while maintaining the lexicographic order as well as an area for all other Unicode scripts.

4.2.3 Variable hash functions for overloaded zones

Enlarging the CAN zone dedicated to Latin data would result in an improved data dissemination among peers, as shown in Figure 4.3. By extending this zone up to CAN coordinate 0.7 on both dimensions (Figure 4.3(b)), it is possible to distribute the load over almost all peers in the overlay. Consequently, the areas dedicated to other unused alphabets shrink. Concretely, this means that each peer being fully included in the highlighted area on these figures becomes exclusively responsible for a given subset of Latin-encoded values. Thus, extending the CAN interval dedicated to a given Unicode subset requires peers in this CAN interval to reduce the Unicode interval they manage. In the overlay, the highlighted CAN interval $[0; 0.7]$ is now associated with the Unicode interval $[A; Z]$. This implies

that, on the horizontal dimension, peer $p1$ becomes only responsible for data between $[A; J[$, $p2$ between $[J; T[$, while $p3$ manages $[T; Z]$ as well as other Unicode scripts.
As this mapping between Unicode and CAN values differs from the standard mapping previously introduced, a *non-standard* hash function must be used to place Latin data items over the enlarged area. The corresponding hash function for the overlay in Figure 4.3 (to be applied on any "enlarged" dimension) is shown in Figure 4.4. Extending the Latin interval in the CAN leads to a different hash function for all peers in the highlighted area, as the end of this interval (represented by the character Z) is now located at CAN coordinate 0.7.



(a) Extension on the horizontal dimension.

(b) Extension on both dimensions.

Figure 4.3 – Possible enlargements of the CAN area for Latin Unicode data storage, in order to disseminate skewed values over several peers while still dedicating a part of the overlay to other Unicode scripts.

Therefore, by changing a hash function, a skewed Unicode interval can be managed by more peers, and data distribution improved among them. As a matter of fact, it is possible to statically define an optimal hash function if data distribution is known beforehand. For instance, if we know we will never have to store non-Latin characters, we can set the Latin interval between pre-defined CAN coordinates, such as $[0; 0.95]$. However, we believe that such system should be able to dynamically modulate its areas devoted to the storage of a specific Unicode range at runtime. For example, after receiving a large dataset of which characters

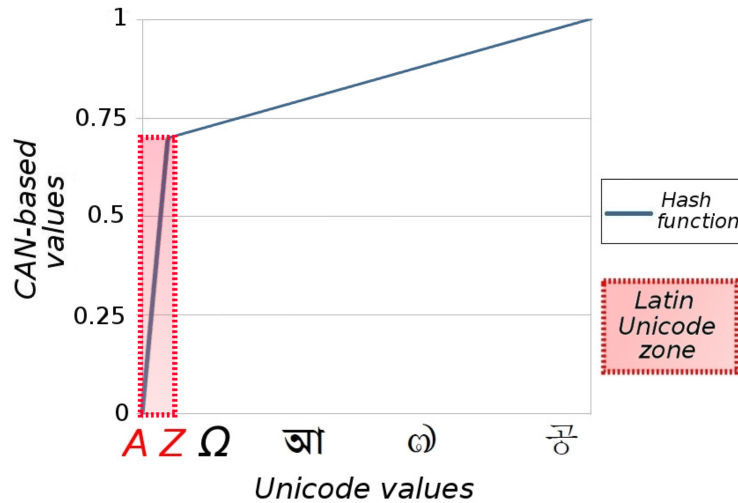
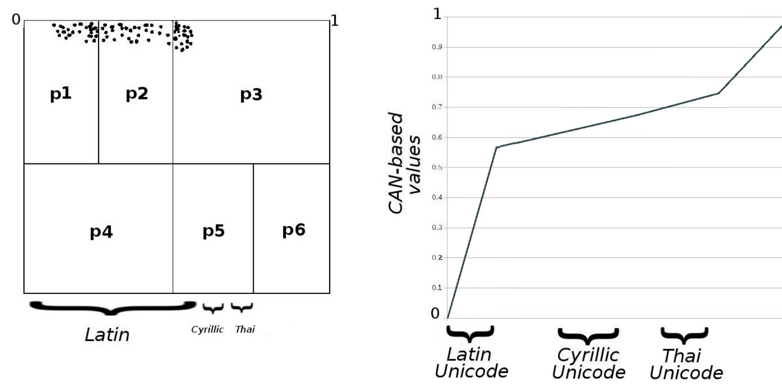


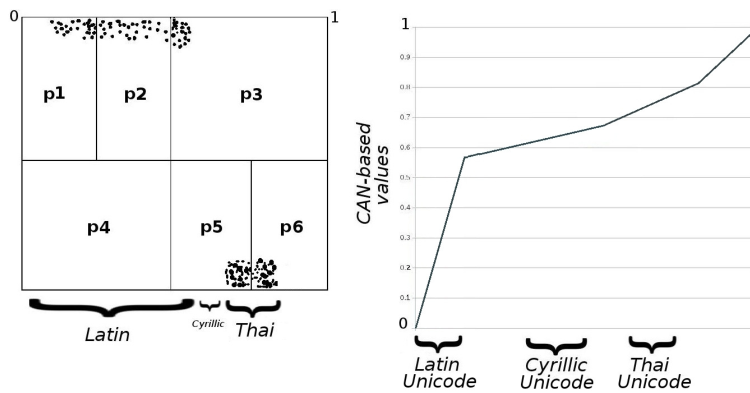
Figure 4.4 – Non-standard hash function to place data items over the enlarged area for skewed values (highlighted on the graph).

are of a specific Unicode script, an enlarged CAN interval for this alphabet will be calculated by the concerned peers, on the basis of some parameters that we will introduce later in this thesis. Generally speaking, any system needing real-time adaptation to incoming data should be able to adapt its hash function at runtime.

In summary, our contribution aims at dynamically adapting the size of areas allocated to skewed values, by changing hash functions to determine where data should be stored. As in any standard structured overlay, all peers initially use the same uniform hash function. When a peer is overloaded because it stores too many skewed values, we believe it should reduce the Unicode interval it is responsible for, in order to store less data. As shown previously, changing this Unicode interval implies for the peer to change the hash function it applies on data. For example, in Figure 4.5(a), new hash functions successively applied by $p1$, $p2$ and $p3$ evenly distribute data load between these peers, by enlarging the Latin interval on the horizontal dimension of the CAN. As there is no non-Latin data to store, there is no need to dedicate large areas in the CAN to other Unicode sets. Thus, these empty areas are shrunk in order to give more space to the currently stored data. If, for instance, large Thai datasets are later inserted, peer $p5$ will have to change



(a) New hash functions applied by p1, p2 and p3 improve distribution of data made of Latin characters.



(b) New adaptation of hash function after Thai data is inserted.

Figure 4.5 – Successive updates on hash functions to improve the skewed data dissemination.

its hash function in order to balance its load (with peer $p6$, in Figure 4.5(b)).

In the next section, we describe the hash function update process and rules that must be followed.

4.3 Hash Function Update Process

In the following, we will describe the process that happens when a peer wants to change its hash function. We will also present some of the constraints that must be taken into consideration, in particular to maintain the same topology (i.e. same neighboring links) and deal with concurrent hash function updates. This will allow us to eventually identify the key rules to observe when using variable hash functions in a CAN overlay.

4.3.1 Update propagation

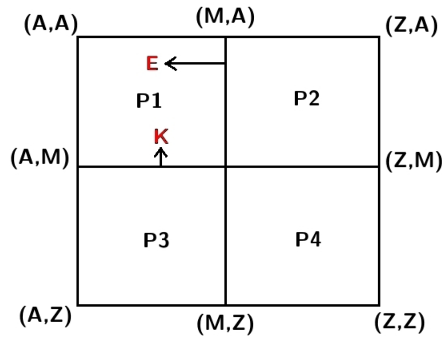
When a peer decides to change its hash function, this corresponds to reducing the Unicode interval it is responsible for. In other words, the peer modifies the Unicode value associated to one of its bounds. The peer first applies the update on itself, then sends a multicast update bound message. This message contains the following information:

- The CAN coordinate of the bound whose Unicode value has been modified. As CAN-based values always refer to the same coordinate in the overlay, this is to ensure other peers receiving this message can immediately locate where the update has been done.
- The new Unicode value associated to this CAN bound, in order for recipients to either apply this new value on their corresponding CAN bound or at least to be aware of this new hash function (we will detail both cases in the following).
- The dimension on which the update has been done, as a peer can use different hash functions on different dimensions.

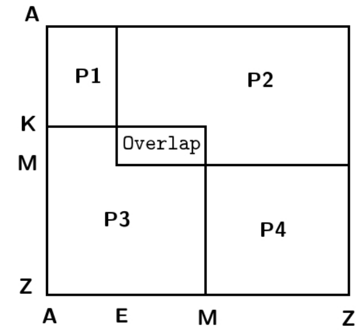
The message is propagated to all concerned peers on this CAN-based bound to make them change their associated Unicode value. Indeed, all other peers having one of their bounds on the same CAN-based value on the same dimension must apply the new value, too. This is very important in order to keep the same topology (same neighbors) and avoid inconsistency within the CAN. As shown in Figure 4.6, if the new Unicode value on a given CAN bound is only propagated to the peer's direct neighbor on this bound, empty zones no one is responsible for, or overlap zones managed by two peers may appear, causing trouble when inserting or looking for a data item.

Figure 4.7 shows the scope of a hash function change in a CAN. Peer $p1$ decides to change its hash function to reduce its Unicode interval on CAN-based bound 0.75. Thus, all peers on 0.75 (the grey area) should also apply this new hash function on 0.75. A multicast message is sent by $p1$ and routed to all concerned peers. Some of them may include the concerned CAN coordinate in their CAN interval while not having any of their CAN bounds precisely at this coordinate, like $p3$, whose CAN interval $[0.5; 1]$ comprises 0.75. As a result, $p3$ will not apply the new hash function but must be reached in order to continue routing the multicast message towards $p4$. But $p3$ is also included in the concerned peers because it must be aware of the new hash function on 0.75, for two reasons. First, $p3$ has to maintain its knowledge about its neighbors $p1$, $p2$, $p4$ and $p5$ up-to-date, like in any standard CAN overlay. In the remaining of this thesis, we consider that each peer knows its neighbors and the hash function they use for each dimension. This is a realistic assumption as this information can be piggybacked on heartbeat messages. Thus, the routing table of a peer can store several hash functions and the peer associated to each of them. The second reason concerns the handling of node arrival. If, later, a new peer wants to join $p3$ and split on its horizontal interval, $p3$ and the new peer will both know which Unicode value to apply on their new 0.75 CAN bound.

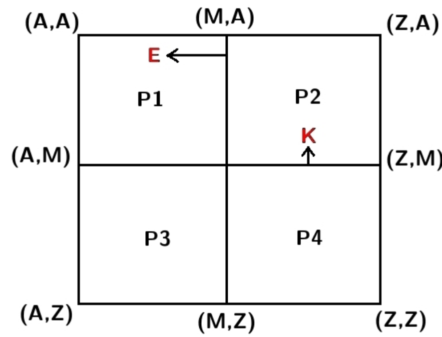
An overloaded peer cannot predict if its hash function change will overload some of the concerned peers since they may not be its neighbors. However, this can be easily addressed, as peers applying this new hash function can also induce



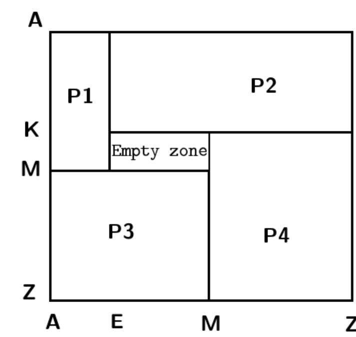
(a) Case 1: P1 successively reduces its Unicode intervals on both dimensions.



(b) P2 and P3 both end up responsible for Unicode intervals $[E;M[$ (horizontally) and $[K;M[$ (vertically).



(c) Case 2: P1 reduces its horizontal interval from $[A;M[$ to $[A;E[$ while P2 reduces its vertical interval from $[A;M[$ to $[A;K[$.



(d) There is no peer responsible for the zone between coordinates (E,K) and (M,M).

Figure 4.6 – Consequences of not spreading the new Unicode value of a CAN bound on a whole dimension (the new value is only propagated to the initiator peer's direct neighbor on this bound).

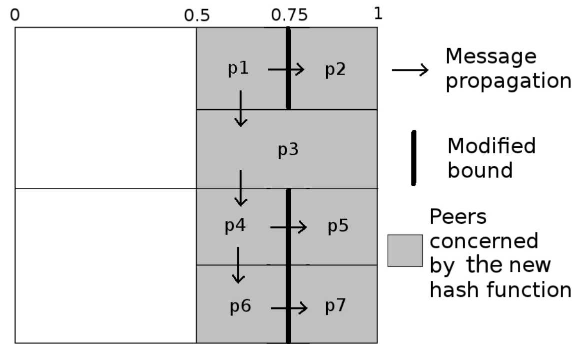


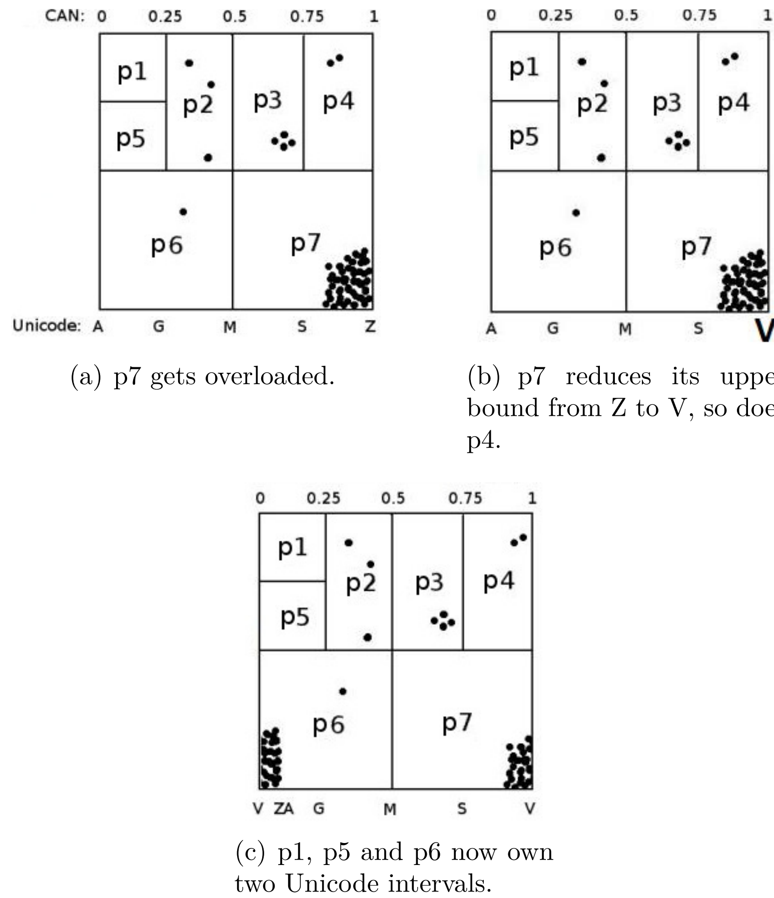
Figure 4.7 – Hash function update message routing.

their own hash function change later on.

4.3.2 Bound reduction in a torus-like topology

Peers having their CAN-based upper bound equal to the maximum authorized CAN value C_{max} should also be able to reduce their upper Unicode bound. Otherwise, if, for example, a large amount of Chinese data items is sent into the system, they would probably be stored by the peer at the far-right of the CAN on each dimension. This would offer no possibility for this peer to extend the CAN interval for these items to the right of the CAN, as they are already nearly located at C_{max}. To avoid this situation, we allow peers located at the far-right of the CAN to reduce their upper Unicode bound like any other peer. To do so, we take advantage of the tore-ring topology of the network, and consider $C_{min} = C_{max}$, which means all Unicode updates associated to C_{max} must also be applied to all peers on C_{min}. As a consequence, peers on C_{min} can become responsible for two Unicode intervals within a single CAN-based interval. This is shown in Figure 4.8(a), representing a CAN where the minimum Unicode value allowed U_{min} is A and the maximum U_{max} is Z. In Figure 4.8(a), $p7$ is overloaded by data located near C_{max}. Although its upper Unicode bound is currently equal to U_{max}, $p7$ is allowed to reduce its value, from Z to V in Figure 4.8(b), in order to send its data between $[V; Z]$ forwards in the overlay (i.e. to $p6$). Consequently, in Figure 4.8(c), peers $p1$, $p5$, and $p6$ own two different intervals: respectively $[[V; Z]; [A; M[]$ for $p6$ and $[[V; Z]; [A; G[]$ for $p1$ and $p5$. Therefore, when a peer is responsible for two Unicode intervals, the

first interval must stop at U_{max} and the second one must start at U_{min}.



(a) p7 gets overloaded.

(b) p7 reduces its upper bound from Z to V, so does p4.

(c) p1, p5 and p6 now own two Unicode intervals.

Figure 4.8 – Unicode reduction process on the C_{max} bound.

4.3.3 Optimal update propagation in a torus-like CAN

In order to efficiently propagate an update bound message, as introduced in Section 4.3.1, an optimal multicast algorithm is used. The goal of such algorithm is to ensure a message will be received only by concerned peers and only once. We based our implementation on the optimal broadcast algorithm for CAN [28] introduced in Chapter 2.3.4. However, two main differences with the original approach occur in our case:

1. We needed to apply a *multicast* algorithm only concerning peers whose CAN

interval includes the modified CAN bound, unlike the original algorithm which is dedicated to *broadcast*.

2. As explained in Section 4.3.2, our CAN uses a torus-like topology, whereas the optimal broadcast algorithm is based on a bounded CAN. This is an important difference as the original algorithm is optimal (a peer receives one and only one message) because routing on a given dimension stops when reaching C_{min} or C_{max} on this dimension.

We describe thereafter the new algorithm implemented to multicast update bound messages in our CAN, and present the main differences with the original broadcast algorithm.

Optimal multicast algorithm

The optimal multicast algorithm presented in Algorithm 4.1 follows the same principles as the optimal broadcast algorithm presented in Algorithm 2.1. However, this time, only peers whose CAN interval includes the modified CAN bound are concerned. Thus, the corner constraint (line 14) must concern the corner including the modified CAN bound. By default, the corner constraint compares the minimum CAN bound of a peer and its neighbor's. This time, if the modified CAN bound corresponds to the maximum CAN bound of the peer processing the message, the corner constraint is checked using the maximum CAN value of peers instead of the minimum. Concerning the spatial constraint (line 26), it is now necessary to check if the neighbor's CAN interval includes the modified CAN bound, otherwise there is no need to route the message. Figure 4.7 shows more concretely how a message propagates among peers when using this algorithm.

To implement this optimal multicast algorithm in an unbounded CAN (torus), no change is necessary as long as the modified CAN bound is not C_{min} nor C_{max}. If it is, a different behavior is required:

1. When routing a message on the dimension of the modified CAN bound, the **getNeighbors** function (line 7) must also include neighbors from the torus perspective. For other dimensions, only neighbors from the bounded perspective are being considered (default behavior).

1: **function** ROUTE_UPDATE_BOUND_MESSAGE($msg, dim, direction$)
2: **if** $direction = forwards$ & $max_{peer}^{dim} = C_{max}$ **then**
3: stop routing
4: **else if** $direction = backwards$ & $min_{peer}^{dim} = C_{min}$ **then**
5: stop routing
6: **else**
7: **for each** $neighbor \in peer$.getNeighbors(dim,$direction$) **do**
8: **if** CHECK_CORNER_CONSTRAINT($neighbor, msg$)
& CHECK_SPATIAL_CONSTRAINT($neighbor, msg$) **then**
9: route(msg, $neighbor$)
10: **end if**
11: **end for each**
12: **end if**
13: **end function**

14: **function** CHECK_CORNER_CONSTRAINT($neighbor$, msg): Boolean
15: **for all** $dim > msg.current_routing_dimension$ **do**
16: **if** $dim = msg.dimension_of_update$ &
$max_{peer}^{dim} = msg.modified_CAN_value$ **then**
17: **if** $max_{neighbor}^{dim} < max_{peer}^{dim}$ **then**
18: **return** $false$
19: **end if**
20: **else**
21: original function behavior
22: **end if**
23: **end for**
24: **return** $true$
25: **end function**

26: **function** CHECK_SPATIAL_CONSTRAINT($neighbor$, msg): Boolean
27: $d \leftarrow msg.dimension_of_update$
28: **if** !($min_{neighbor}^{d} \leq msg.modified_CAN_value \leq max_{neighbor}^{d}$) **then**
29: **return** $false$
30: **else**
31: **for all** $dim < msg.current_routing_dimension$ **do**
32: original function behavior
33: **end for**
34: **end if**
35: **return** $true$
36: **end function**

Algorithm 4.1 – Optimal multicast algorithm. In this algorithm, *peer* refers to the current peer processing the message.

2. In order to avoid loops and also to keep the algorithm optimal, once a message has crossed C_{min} or C_{max} (the one that is modified), the recipient must stop routing on this dimension. Otherwise, routing might continue indefinitely: let us consider two peers $p1$ and $p2$ respectively located on CAN intervals $[0; 0.5[$ and $[0.5; 1]$ on dimension 0. If $p2$ modified the Unicode value associated to CAN value 1, $p2$ would route a message on dimension 0 asking $p1$ to update its CAN bound 0. Then, as $p2$ would match all constraints from the optimal multicast algorithm, $p1$ would forward the update message to $p2$, and so on.

Obviously, such technique could also be used more generally to implement optimal multicast for other purposes, either in a bounded or a torus CAN. This algorithm allowed us to use optimal multicasting in our system, to avoid duplicate communication messages and unnecessary broadcast.
In the next subsection, we will go further into our wish to limit the communication between peers when balancing the load.

4.3.4 Concurrency and message filtering

For a given bound, a peer p receiving an update message will allow the new Unicode value only if it is lower than p's current Unicode value for the same CAN-based bound. Thus, it is forbidden to ask for a forwards reduction of the minimum bound, in order to avoid inconsistent concurrent reductions (i.e. on different directions). For example, in Figure 4.9, if two peers ($p1$ and $p4$) decided in a short time interval to modify their Unicode value (respectively $Umax_{p1}$ and $Umin_{p4}$) for the same CAN-based value (0.5) but on different directions (backwards for $p1$ and forwards for $p4$), it would become difficult to tell which one has priority over the other, especially since their multicast messages may not be received in the same order by all the concerned peers ($p1$, $p2$, $p3$, $p4$). Therefore, only $p1$ should be allowed to reduce its bound.
For even more safety regarding concurrent updates, we do not allow a peer to modify its hash function while the peer has received update messages it has not processed yet. Moreover, if update messages asking for different Unicode values are sent by different peers at the same time and for the same CAN-based bound,

we ensure all peers will end up choosing the same value: the lowest one. Thus, peers will reject the other messages and stop their multicast routing. The only case where a higher Unicode value would be accepted is when the reduction is made by a peer already containing two continuous intervals on one dimension (as discussed above in 4.3.2) and the new value is included in the first interval. In Figure 4.10(a), if, later, $p1$ wants to reduce from G to Y, it would no longer be responsible for two Unicode intervals (only $[V;Y[$), but its neighbor $p2$ would ($[[Y;Z];[A;M[]$), as shown in Figure 4.10(b).

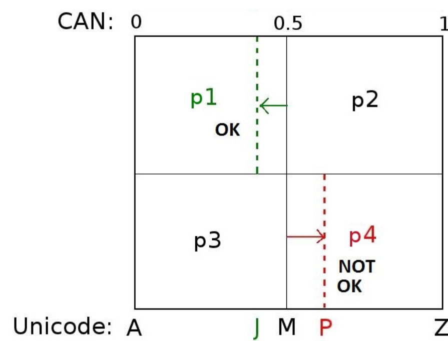
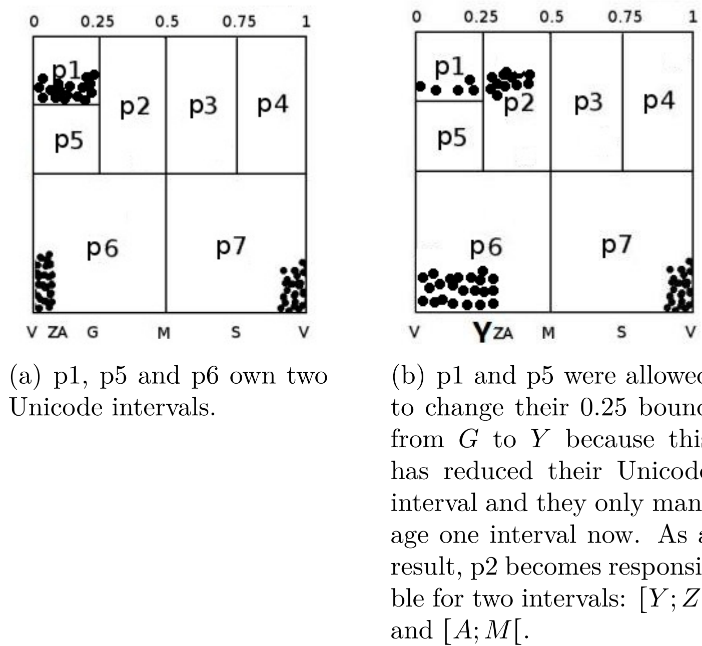


Figure 4.9 – Inconsistent reductions for the same CAN-based value (0.5).

Our rule selecting the lowest value is very efficient as it guarantees the same value will be picked by all concerned peers. The optimal multicast algorithm ensures that all peers having to apply the new Unicode value will receive the message. Even if several peers initiated different update messages for the same CAN bound in a short period of time, and no matter in which order the concerned peers would receive the multicast message, all peers on this CAN bound will use the same Unicode value after all messages with the lowest value have been processed. The main benefit of such rule is that peers independently take the decision to apply or reject a new bound value but they all end up choosing the same value, without any mutual consultation. This solution is more convenient than those based on a two-phase commit-like approach, such as mutual exclusion algorithms [66], using lock mechanisms. Besides, these algorithms require the initial sender to know how many peers are concerned by the message, in order to know how many replies should be received by the sender prior to any action. The process ends when all



(a) p1, p5 and p6 own two Unicode intervals.

(b) p1 and p5 were allowed to change their 0.25 bound from G to Y because this has reduced their Unicode interval and they only manage one interval now. As a result, p2 becomes responsible for two intervals: $[Y; Z]$ and $[A; M[$.

Figure 4.10 – Bound reduction on peers responsible for two Unicode intervals.

nodes have applied the update and notified the initiator. This is very difficult to implement on a P2P overlay such as CAN, because peers have no overall knowledge of the system. Thus, a peer cannot estimate how many peers are also located on the same CAN bound for the whole network.

During the bound update process, peers on a same CAN bound may use different Unicode values. In the next subsection, we show how this does not affect data redistribution and lookup among nodes.

4.3.5 Data movement and lookup

After a peer p has changed its hash function, it has to send all data it is no longer responsible for to its neighbor(s). To do so, p has to wait until they apply the update as well (it should be done quickly as they are only one hop away from p). After sending its triples, p waits until it receives a storage acknowledgement message from its neighbor(s) before deleting triples, in order to ensure no triple is lost during this process.

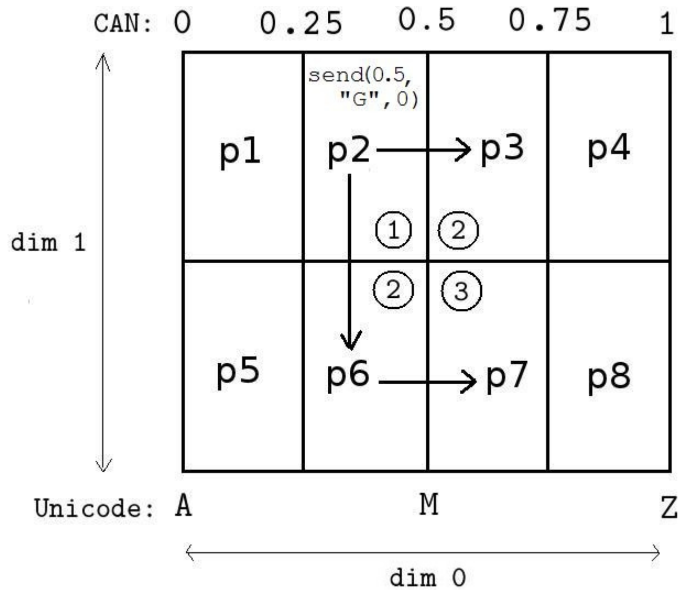
As there is no coordination between peers and no mutual exclusion algorithm is used, all peers on the same CAN bound may not apply the new Unicode value at the same time. However, ultimately, all peers should use the same value. In the meantime, this does not affect data lookup or insertion. Indeed, a peer forwards a message once it has applied the update itself. Therefore, the optimal multicast algorithm ensures a peer will receive an update bound message from a neighbor that already applies the new value, which eases data movement between these peers.

Figure 4.11(a) presents the routing paths followed during an update propagation. First, $p2$ applies the value G on its 0.5 CAN bound on dimension 0. Then, $p2$ emits the update message towards $p3$ on dimension 0 and $p6$ on dimension 1, following the optimal multicast algorithm constraints. Then, these peers apply the update and try to forward the message as well. Peer $p3$ has received the message on dimension 0 which means it cannot forward on a dimension higher than 0 and its neighbor $p4$ is not concerned by the update, whereas $p6$ can forward to $p7$. Therefore, the configuration proposed in Figure 4.11(b) will never happen. Indeed, if the optimal multicast algorithm is used, there is no way for $p7$ to receive, and hence apply, the update message before $p6$. This is because $p6$ is the only peer that can forward the message to $p7$, according to the rules of this algorithm. Thus, $p6$ would necessarily already use "G" on 0.5 before $p7$. This considerably enhances data transfer between peers and the overall consistency of the overlay.
In Figure 4.11(c), we focus on queries management during the update process on dimension 0. We consider that $p2$ currently uses value G on 0.5 whereas its neighbor $p3$ has not received or applied the update yet and still uses value M as $Umin^0_{p3}$. Several cases may occur if one of them receives a lookup/insertion message for a data item:

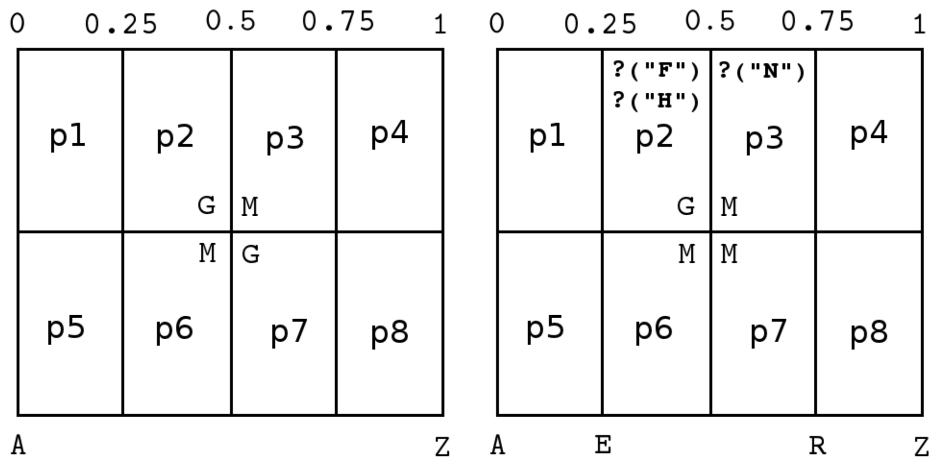
1. If the item value is comprised between $[M;R[$, $p3$ will obviously receive and process the message.

2. If the value is between $[E;G[$, $p2$ will be responsible for the message.

3. If the value is comprised between $[G;M[$:



(a) Update Bound Message (UBM) propagation.



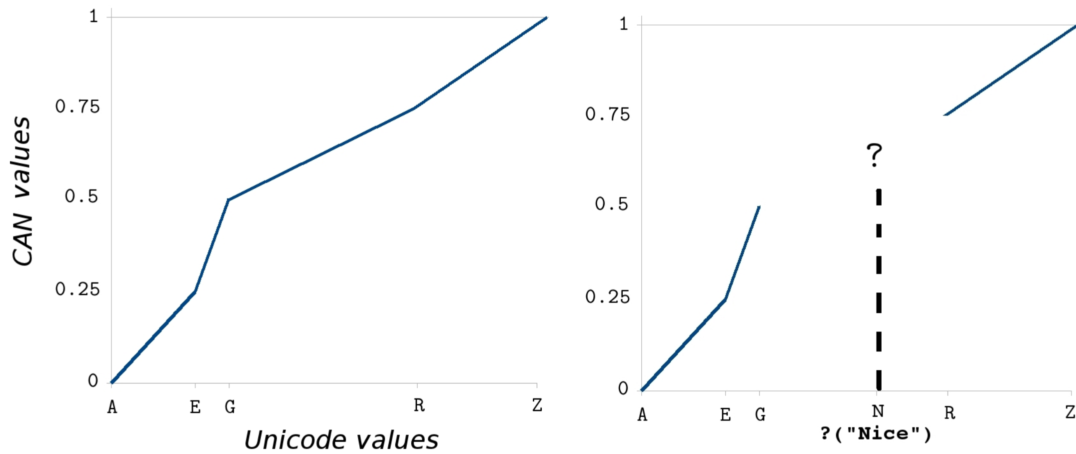
(b) Impossible configuration.

(c) Possible intermediate configuration showing which peer would be responsible for processing which lookup query (denoted as ?(”*item*”)) when using different Unicode bounds, as peers $p3$, $p6$ and $p7$ have not applied the update yet.

Figure 4.11 – Propagation of a new Unicode value and its possible consequences on query processing.

(a) If $p2$ receives the message: as $p2$ still stores data between $[G; M[$ because $p3$ still hasn't told $p2$ that it uses the new hash function, $p2$ should process the query as its neighbor on the right on dimension 0 (i.e. $p3$) still appears as managing data from the M value.

(b) If $p3$ receives the message: $p3$ still uses M on $Umin^0_{p3}$, hence $p3$ considers lower values must be located on the backwards direction on dimension 0. Thus, $p3$ will forward the query to $p2$, which is the right decision.



(a) Overall hash function used on dimension 0. (b) $p1$'s hash function knowledge.

Figure 4.12 – Overall hash function knowledge of $p1$ associated to the overlay presented in Figure 4.11(c).

Using variable hash functions implies peers may not know where exactly in the overlay an item should be located. Let us consider $p1$ (from Figure 4.11(c)) initiates a lookup query to retrieve a data item containing the value "*Nice*" on dimension 0. Figure 4.12(a) represents the overall non-linear function used in the overlay, while Figure 4.12(b) shows $p1$'s knowledge about this hash function. As stated in 4.3.6, $p1$ knows the hash functions used by its neighbors $p2$ and $p4$ (torus). However, $p1$ has no idea about the hash function used between CAN coordinates 0.5 and 0.75 on dimension 0. As $p1$ cannot know to which CAN coordinates "*Nice*" is associated, $p1$ will route its query towards the closest coordinate to "*Nice*" it knows of. In our case, this corresponds to the value R associated to CAN value 0.75 on neighbor $p4$. Then, $p4$, whose hash function knowledge includes $p3$, will

be able to estimate item "$Nice$" should be stored by $p3$ and will route the query to $p3$.

4.3.6 Main rules

On the basis of what was described throughout this section, we were able to draw from all the constraints that we introduced some rules that summarize our approach and that must be followed to use variable hash functions in a CAN overlay.

As there is no central coordination in an overlay, it is very difficult to change the overall hash function applied in the network. However, it is not necessary for all peers to use the same hash function or to know all the different hash functions used in the overlay to get a fully functional CAN. Peers can use different functions as long as the following rules are observed:

(a) A peer should know the hash function of its neighbors for each dimension, and should notify them as soon as it changes its hash function.

(b) Neighboring peers using different hash functions should have the same Unicode value on their common bound.

(c) All peers on a given CAN-based bound should apply the same hash function on this bound. Peers whose CAN interval includes this bound should also know this hash function, without needing to apply it.

(d) When overloaded, a peer is only allowed to reduce the maximum Unicode value it is responsible for.

Rule (a) is required to ensure the arrival of new peers (division of an existing zone between two peers) is correctly handled. Also, when routing data in the network, this allows a peer to choose among its neighbors which one will be the most efficient for routing a particular data using the shortest path. Finally, this rule helps an overloaded peer know when it can start sending part of its data to its neighbor(s), i.e. when both of them apply the same new Unicode value on their common bound.
Rule (b) ensures overall consistency of the overlay: two peers sharing a common CAN-based value should share a common Unicode-based value, even if they use

different hash functions. Given two neighbor peers $p1$ and $p2$ using $h1$ and $h2$ as their respective hash functions on dimension d, if $h1(Umax^d_{p1}) = h2(Umin^d_{p2})$ then $Umax^d_{p1} = Umin^d_{p2}$. For example, in Figure 4.11(c), $p1$ and $p2$ use two different hash functions $h1$ and $h2$ on dimension 0 (depicted in Figure 4.12(a)), respectively between CAN coordinates $[0; 0.25[$ and $[0.25; 0.5[$. However, they share the same Unicode value (E) on their common bound (0.25), i.e. $h1(E)$ and $h2(E)$ both correspond to coordinate 0.25.
Rule (c) keeps in line with rules (a) and (b) and prevents updates on hash functions from creating overlap or empty areas in the CAN. It is also necessary to maintain optimal routing in the overlay. Figure 4.13 describes what would happen otherwise, if a peer modifying the Unicode value of one of its bounds does not propagate this update. This figure considers a CAN where $p1$ modifies its 0.25 bound on dimension 0 from E to C but this new value is not propagated on dimension 1 to $p5$ and $p6$ (only to $p2$, with whom $p1$ shares this bound). The same principle happens when $p2$ and $p3$ also decide to reduce their intervals ($[C; F[$ for $p2$ and $[F; H[$ for $p3$). When $p9$ sends a lookup query for item $< London; City >$ ($London$ being associated to dimension 0 and $City$ to dimension 1), if routing starts on dimension 0, then a long path has to be followed. Indeed, as $p9$ uses R on min^0_{p9}, it will continue routing the query backwards in order to reach values starting by L, so will $p7$. When the query arrives on $p6$, as its interval $[E; M[$ on dimension 0 encompasses $London$ but its interval $[M; Z]$ on dimension 1 does not include $City$, $p6$ stops routing on dimension 0 and starts routing on dimension 1. Thus, $p2$ receives the query that matches its interval on dimension 1 but not on dimension 0, as $p2$ has modified its hash function. Hence, $p2$ restarts routing on dimension 0, until the query arrives on $p4$, that is responsible for the desired item. In contrast, if the new hash functions are propagated to all peers on the modified bound, $p9$ would use H on min^0_{p9}, thus it would not route the query on dimension 0 and, in the end, the routing path would require two hops instead of five[3].

Finally, rule (d) is mandatory to maintain the overlay consistent in case of concurrent updates on hash functions made by different peers. As we explained in subsection 4.3.4, this rule helps peers choose the right new Unicode value for their

[3]One or several more hops may still be necessary until all the peers concerned by the bound modification have processed the update.

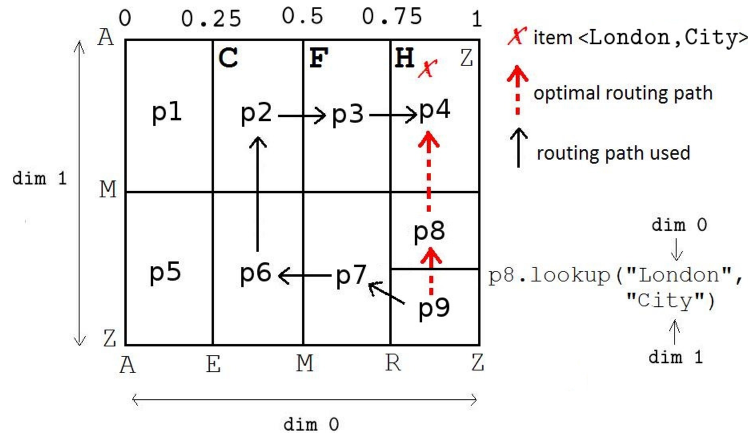


Figure 4.13 – Possible routing path associated to an item lookup when an update is not propagated through the whole dimension.

bound and ensures all concerned peers will end up using the same hash function on a given CAN bound, without any additional consultation message. This rule is specific to the CAN architecture as peers manage a set of coordinates bounded by a minimum and a maximum value, which means a peer could also reduce its Unicode interval by increasing the minimum Unicode value it is responsible for. On other P2P overlays, such as Chord, this rule would not be necessary as a peer is only aware of the maximum value it can manage.

4.4 Summary

In this chapter, we have presented our load balancing strategy for improving skewed data distribution among nodes. Hash functions are at the heart of data distribution for DHT-based overlays. However, few load balancing strategies precisely tackle the issue of hash functions in order to improve data dissemination. Moreover, these strategies aim at replicating data or destroy the natural ordering of data. They do not directly address the skewed data problem whereas real world datasets are known to be highly skewed [67] [10] [68]. Our approach is meant to perform skewed data dissemination without *a priori* knowledge about data distribution.

In most P2P systems, peers do not have an overall knowledge of the overlay. We have shown it is not necessary for all peers to use the same hash function, while continuing to maintain optimal routing and consistency in the overlay. Peers do not even need to be aware of the overall hash function used in the network. To do so, few rules are needed to ensure peers receive enough information about changes while being able to make right decisions without any prior communication with other peers.

When a peer is overloaded, we allow it to reduce the interval it is responsible for by modifying the hash function it applies on data, in order to send part of its data to its neighbor(s). In the meantime, an optimal multicast algorithm, presented in 4.3.3, is used to propagate this new hash function only towards peers concerned by this update (Section 4.3.1). Recipients independently make the decision to accept or reject this update. However, they all end up opting for the same decision, even in the case of multiple concurrent updates (Section 4.3.4). Thanks to such rules, we can easily limit communication between peers while maintaining the overlay consistent. Each peer applies its own hash function on data when routing a lookup query. Therefore, evaluating an item's location is done on the go: each peer routing the message applies its own hash function on the item, which addresses the lack of knowledge of peers regarding the overall hash function of the overlay.

In order to evaluate our strategy, we had to implement the approach presented in this chapter in the context of a distributed storage system. To this aim, we developed a generic API for simplifying coding and maintenance by enabling the integration of various strategies in a system, with minimal impact on the existing business code.

Chapter 5

Generic API for Implementing Load Balancing

Contents

5.1 Motivation **81**

5.2 Load Balancing Differentiators **83**

5.3 API Presentation **88**

5.3.1 High-level abstractions 88

5.3.2 Core API 89

5.4 Case Studies **92**

5.4.1 Big Data management systems 93

5.4.2 P2P systems 94

5.5 Summary **97**

5.1 Motivation

Distributed systems for Big Data management very often face the problem of load imbalance among nodes, especially regarding data distribution. To address this issue, there exist almost as many load balancing strategies as there are different

systems. Many load balancing strategies have been proposed based on replication or relocation. The model followed by these strategies usually consists in controlling resources and/or nodes location. However, many variants are conceivable based on indirection, identifiers, range space reassignation or virtual peers[1]. Moreover, designing a load balancing solution requires to consider parameters such as the overload criteria to take into account, how an overload is detected, and how load information is exchanged. This variety of parameters has led to the definition of multiple solutions that often differ by minor but subtle changes.

When designing a scalable distributed system geared towards handling large amounts of information, it is often not so easy to anticipate which kind of strategy will be the most efficient to maintain adequate performance. This may refer to performance concerning users (e.g., query response time) or the system (scalability, reliability or even node failure prevention). Based on this observation, we present in this chapter the main concepts behind a generic API for load balancing in distributed systems. We describe the methodology behind the building of this API to implement and experiment any strategy independently from the rest of the code. Our contribution simplifies coding and maintenance by enabling the integration of different strategies in a system, with minimal impact on the existing business code. More precisely, we will focus on the context of data management systems. However, the general ideas presented in this chapter can be applied on other types of truly distributed systems. To this aim, we provide a guide of what criteria are important to define and the essential principles to think about before implementing a load balancing strategy.

We propose to decompose into components the main features arising from a load balancing mechanism. This enables changing only a part of a strategy without having to impact the other components. Our contribution is to provide a synthetic

[1]Unlike traditional P2P networks where one peer is deployed per node, virtual peers are an abstraction allowing several peers to be hosted on a same physical node. Upon the detection of an underloaded or overloaded peer, virtual peers are reassigned to other nodes in order to maintain the machine load under a given threshold. Some strategies based on virtual peers were presented in Section 3.2.2. Other solutions relying on some of the other aforementioned concepts can be found in Chapter 3.

and operational vision of how the different bricks that form a load balancing strategy articulate, and at which time in its life cycle these bricks are used. Therefore, our solution is especially useful when it becomes necessary to experiment with various load balancing strategies in a system, prior to a definitive choice for instance.

In the following, we introduce differentiation criteria that we consider as the main elements composing a load balancing strategy. Then, we present our common API to implement any kind of strategy. Finally, we show how this API is compatible with some relevant load balancing strategies presented in Chapter 3, either from the literature or used by famous Big Data systems. This work was done in collaboration with Laurent Pellegrino.

5.2 Load Balancing Differentiators

By studying the state of the art of load balancing in distributed systems, we identified the main differences between each strategy, in order to determine the right level of abstraction to be able to implement them all. Although they seem very different and are applied on various types of distributed systems, all the load balancing strategies previously cited and most other existing solutions rely on the same principle. A node moves a given amount of *load* to a certain *target* which will become responsible for the *load* being moved. The decision to move load always comes after a load comparison with a given *source* of information. It is very common to trigger this load comparison during a specific state of the system such as network construction, data insertion or periodically. Overall, we identified the following differentiators to establish a load balancing strategy. They represent the main concerns to focus on in order to develop a strategy.

(a) **Criteria**

Before fixing load imbalances, a disproportion in terms of load must be detected. This implies to know which load criteria are involved and how their variation is measured on nodes. This differentiator defines which load variations are considered and to which resource(s) (CPU, bandwidth or disk usage) and operation(s) (e.g., item lookup, item insertion, response latency for a user

request, etc.) they refer. Usually, few criteria, not to say only one, are taken into consideration in practice.

(b) **Load State Estimation Algorithm**

This step consists in defining whether a node is experiencing an imbalance or not, and how this decision is made. Usually, a node relies on a source of load information containing aggregated remote information (see differentiatior (g) to figure out how this source of information is populated with remote information) or uses purely internal information by comparing its internal load(s) with predefined threshold(s). Thus, this estimation algorithm can also implicitly specify the performance requirements (for instance, in terms of SLAs) of a particular system for any criterion defined in differentiator (a).

(c) **Load Balancing Decision**

The decision to trigger load balancing often differs from a load balancing strategy to another. This differentiator aims at identifying who triggers the evaluation of the system/a node's load state and when. Consequently, it is related to the time at which the whole load balancing mechanism is triggered by a node and will necessarily impact how a load balancing implementation is welded to an existing system's business code.

(d) **Load Balancing Mechanism**

The mechanism identifies which well-known solution is applied to move load from a node to its *target*. As summarized in the introduction of this chapter, it may consist in using virtual peers, redirection pointers or even range space reassignation. It helps checking whether prerequisite abstractions, required to define a given load balancing strategy, are available or not.

(e) **Load to Move**

Once an imbalance is detected, the next stage is to fix it. This implies to know what is the load to move. This differentiator defines the amount of load to move from a node to its *target* but also how this load is selected.

(f) **Target**

Given an unbalanced node, its target is a set of nodes used to balance its load with. In other words, it describes who receives the load when the load balancing process is triggered.

(g) **Load Information Exchange**

A strategy optionally embeds a mechanism to exchange information. It is often used to compare the internal load with an average system load estimated through exchanged information. This differentiator defines when the estimations are transferred (if they are), from who and how. Once received on a node, these estimations compose a *source* of information.

(h) **Load Information Recipients**

Given a node n, the recipients are nodes that share load information with n. They are mainly used to build a *source* of information involved in the load balancing decision process.

Tables 5.1 and 5.2 present how some of the strategies presented in Chapters 3 and 4, respectively applied by Big Data and P2P systems, match our differentiators. It is interesting to note that systems for Big Data storage and processing, like MongoDB and AWS, usually allow to customize very few of these differentiators. The strategy applied in MongoDB may vary the behavior of the *Load State Estimation Algorithm* and the periodicity of the *Load Balancing Decision*. With AWS Auto Scaling, the periodicity of the *Load Information Exchange* and the way to evaluate the load state of an instance (*Load State Estimation Algorithm*) based on some load *Criteria* may change. Such strategies are effective in one specific case but may no longer be as efficient after any change that might occur, for example after a system update or to meet a fast-growing user demand. In such cases, it may be complicated to modify a load balancing strategy if it is hard-coded. This is why we propose a generic API that eases the development of load balancing policies in distributed systems.

	MongoDB (NoSQL DB)	**Nasir *et al.* (stream)**	**AWS (Cloud)**
Criteria	Number of chunks stored by a shard	Number of messages sent by each source to a worker	Some metrics among the following: CPU usage, latency, number of I/O operations, ...
Load State Estimation Algorithm	Triggering rebalancing after a new shard has been added or if new chunks of data have been inserted, causing an imbalance among shards that exceeds the migration threshold	Always triggering rebalancing when a source produces a new message	Whenever a metric exceeds a predefined threshold
Load Balancing Decision	Periodically, on each router	When a source sends a message	Periodically, depending on CloudWatch's granularity to retrieve metrics
Load Balancing Method	Sharding (horizontal scaling)	Power of two choices paradigm	Registration of a new instance (if overload), termination of an existing one (underload)
Load to Move	Part of the heaviest shard's chunks	The message sent by the source	Part of the application's future requests/traffic
Target	The lightest shard in the system	The least loaded worker among those selected by the random hash functions	A new instance (if overload) or an existing instance (if underload)
Load Information Exchange	Periodic probes (*pull*) from the balancer process of routers to Config Servers and *push* calls from a shard to Config Servers whenever the shard stores a new chunk	Internal to the source: the source checks its local registry to retrieve the amount of load it has previously sent to the workers matching the hash functions	Instances monitoring (*pull*) from Amazon CloudWatch
Load Information Recipients	Config Servers	None	All instances

Table 5.1 – Load balancing strategies for Big Data systems mapped to the differentiators.

	Rao *et al.*	**Meghdoot**	**Byers *et al.***
Criteria	Resource agnostic (storage, bandwidth or CPU) but only one criterion	Subscriptions and data popularity	Number of data items per peer
Load State Estimation Algorithm	Given L_i the load of node i (sum of the load of all the virtual servers of node i) and T_i a target load chosen beforehand, a node is heavily loaded if $L_i > T_i$, lightly loaded otherwise	Always triggering rebalancing when a new peer joins the system	Always triggering rebalancing when receiving an item to insert
Load Balancing Decision	Periodically, on each peer	When a peer joins the system	Upon the insertion of an item (data) on the entity that performs the insertion
Load Balancing Method	Virtual peers transfer (with no virtual peer split or merge)	Range space reassignation or replication (zone + subscriptions)	Power of two choices paradigm
Load to Move	One of the overloaded peer's virtual server	Half of a heavy peer's subscriptions or replication of a heavily loaded peer	The item to be inserted
Target	A random peer, an underloaded peer or the best underloaded peer (depending on the scheme)	The heaviest peer in the system known by the new peer joining the system	The least loaded peer among those contacted for a given item
Load Information Exchange	Random probing for the first scheme (*pull*). Periodic load advertisement from lightly loaded peers (*push*) and sampling from heavily loaded peers (*pull*) with the second scheme. The third scheme implies information exchange from peers to a directory (*push*)	Periodically, peers update neighbors about their load (*push*) and share their list that contains the k most heavily loaded peers detected	The peer that wants to insert an item computes d hash values and contacts the associated peers to retrieve their load (*pull*)
Load Information Recipients	A peer managing a random id for the first scheme. A directory node for the second and third schemes	One hop neighbors	For d hash functions applied on an item to insert, the n peers managing the computed hash values

Table 5.2 – Load balancing strategies for P2P systems mapped to the differentiators.

5.3 API Presentation

From the identification of the differentiators presented in the previous section, we were able to determine the key requirements for our API. This allowed us to define the essential methods to model any strategy. In this section, we define the components and functions of our API, based on the differentiators presented above. An approach based on hierarchical components was deliberately used because components enable modularity and cohesion [69], which eases reusability.

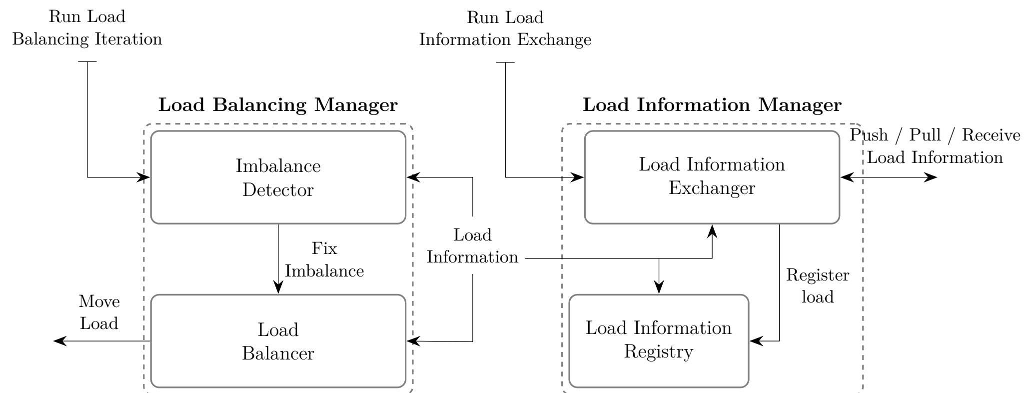


Figure 5.1 – Basic abstractions for a generic API.

5.3.1 High-level abstractions

The features associated with differentiators (a) to (f) relate to the management of load balancing and could be gathered in a so-called *Load Balancing Manager* component. By pushing our analysis deeper, we may argue that differentiators (b), (c) and (e), (f) identify two separate subcomponents. Indeed, the first group of differentiators relates to the detection of imbalances (*Imbalance Detector*), whereas the second (*Load Balancer*) captures the method and the information required to balance the load in case of imbalance. Finally, differentiators (g) to (h) are merely involved in the process to give feedback about the resource utilization per criterion to nodes. In a component-oriented approach, this could be modelized as a *Load Information Manager* with a subcomponent, dubbed *Load Information Exchanger*, in charge of exchanging load information.

Figure 5.1 illustrates these components in charge of isolating these load balancing features on each node. In addition to the components presented above, the figure sketches an additional one, named *Load Information Registry*, that aims at linking the two main composite components (*Load Balancing Manager* and *Load Information Manager*), since each may run in its own flow of control. The components in Figure 5.1 represent the whole load balancing abstraction covered by our API. However, depending on the system's architecture and the chosen strategy, all the nodes do not necessarily have to implement all these components. One such example will be given in Section 5.4.2.

Components are wired together by calling actions on other components. Some actions carry *load information* which contains the following values:

- *node*: the node sending its *load information*. Can be a node identifier, a reference, etc.

- *criterion*: type of *load* (disk space, CPU consumption, bandwidth, etc.).

- *load*: load of the *node* for a given *criterion*.

These attributes and their value can be expressed in the form of a key/value list. Optional elements such as an *optimal load*, *internal threshold* or *timestamp* can also be included. Details about internal components actions and their behavior are given in the next subsection.

5.3.2 Core API

The function calls defined below capture the core of load balancing strategies, classified per component. The signature for the required functions is given in a simple untyped pseudo language, thus allowing any particular implementation. We describe in the following the usefulness of each method of our API. However, the way each function is implemented is totally application dependent. Before entering into the description of the API for each simple component, it is worth noting that the two main composite components identified previously respectively expose a **run_load_balancing_iteration()** and a **run_load_information_exchange()** function. They act as entry points for node instances to execute one step of the

two complementary composite components code, thus orchestrating in which order the functions introduced below are run.

Load Information Exchanger

This component is responsible for sending the node's *load information* and receiving *load information* from other nodes in the network.

- **exchange_load_information**$(recipients,\ load_information)$

  $\rightarrow load_information$

A node sends and receives *load information* from other nodes, for a given load *criterion* (storage, CPU, etc.) and a corresponding amount of *load*. The **exchange_load_information** function may return *load information* from pull calls or periodically sent by other nodes, that will be directly used by the *Load Balancer* component or stored in the node's *Load Information Registry* (see details below). A push call is used when a node wants to unilaterally notify *recipients* (a given number of nodes: neighbors, all nodes, a random node, etc.) about its load state (*load information*). In some systems, like Cloud infrastructures, pull calls may only be executed by load balancer nodes monitoring the other instances.

Load Information Registry

This registry stores all *load information* received by a node. Optionally, time can be taken into account when storing information as it is possible to maintain synchronous clocks using protocols such as NTP.

- **register**$(load_information)$
- **get_load_report**$(criterion,\ nodes) \rightarrow load_information$

The **register** function writes into the registry *load information* received by the node's *Load Information Exchanger*. The **get_load_report** operation provides *load information* for a given set of *nodes* according to a certain *criterion*. With this method, the calling node can retrieve relevant information about either its own load state only, or about the load state of other nodes. A node can also use this function

to get estimates, averages and other statistics based on the information contained in its registry. These estimates are calculated thanks to the load information messages received and stored earlier. The returned *load information* can help estimate the overall average load or the load of a given node, for example. There can be no result if the calling node has not recently received any *load information* message from the concerned *node(s)*.

Imbalance Detector

The default behavior is to check if a load criterion is unbalanced (overload or underload), in order to trigger a load balancing strategy.

- **make_decision**(*criterion*) → *load_state*

Using a given algorithm, this function determines whether to induce a load balancing strategy or not, according to a given *criterion.* This operation is basically meant to return an enumerated type: *overloaded* or *underloaded* if a rebalance is necessary, *normal* otherwise. The returned value may depend on a threshold value or not, typically to detect overload or underload. If a threshold value is used, it can be calculated using *load information* provided by the *Load Information Manager* (locally, from the *Load Information Registry* using **get_load_report**, or remotely by contacting nodes with **exchange_load_information**). However, depending on the strategy being used, it is not necessary that all nodes implement **make_decision**, for example if a super node retrieves the load state of other nodes and makes the decision to move workloads among them, which is typical for systems like Cloud infrastructures.

Load Balancer

This component is responsible for balancing the load.

- **select_load_to_move**(*load_information_manager*, *criterion*, *load_state*) → *load_to_move*
- **select_target**(*load_information_manager*, *criterion*, *load_state*) → *target*

- **rebalance**(*criterion*, *target*, *load_to_move*)

The **select_load_to_move** operation returns the amount of *load to move* from one node to another. Optionally, it is possible to use local or remote information from the *Load Information Manager* to determine how much load has to be moved. As moving load throughout the network may impact system performance, some systems may also call for their own cost function in the **select_load_to_move** implementation. This way, it would be possible to evaluate whether moving load would be counterproductive or not, and if not, what would be the right amount of load to move to maintain a system's own performance requirements.
The **select_target** function is responsible for finding which node(s) will receive this *load to move*. To do so, it is possible to query the *Load Information Manager* but it is not mandatory (*target* can be a random node, a new node, etc.).
Finally, **rebalance** is the only method used to concretely move some load between a node and the target. This method can be implemented with any particular behavior defined by a load balancing strategy to describe the migration process in itself as well as constraints to be observed during data migration. For instance, **rebalance** may also include the instructions to be followed concerning data consistency during the migration between nodes, in order to avoid read/write conflicts, data loss or the creation of replicas between the two nodes involved in the data transfer.

5.4 Case Studies

Many papers describe load balancing solutions for distributed systems, using various strategies. In the following, we focus on six different systems (three based on P2P and three in the context of Big Data storage) presented in Chapter 3, each implementing its own load balancing strategy. Although the chosen papers/systems do not constitute an exhaustive list of load balancing solutions, they are representative of existing works. Indeed, these strategies are applied on various distributed systems used in different contexts (P2P for data storage or publish/subscribe, NoSQL database, stream processing, Cloud computing). Load balancing

is triggered at different states: periodically, after a new node has joined the system (to implement horizontal elasticity for instance), when inserting data, or to meet user service level expectations. Besides, and perhaps more importantly, these strategies are either from papers among the most-cited for the topic, or pertain to some of the most famous distributed infrastructures for data storage and processing (MongoDB, Apache Storm, Amazon EC2). In the following, and based on the taxonomy identified earlier in Tables 5.1 and 5.2, we describe how our API could prove successful in implementing these strategies, although very different at first sight.

5.4.1 Big Data management systems

MongoDB new shard insertion

Routers use a balancer process that periodically retrieves from Config Servers *load information* about shards (*Pull Load Information* via *Load Information Exchanger*). Config Servers provide such information thanks to *Push Load Information* calls sent by each shard every time it becomes responsible for a new chunk. After a new shard has been added to MongoDB, a router will detect, thanks to information retrieved by its *Load Information Exchanger*, an uneven chunk distribution and will trigger load balancing (*Imbalance Detector*). The router's *Load Balancer* will then calculate the amount of chunks that should be removed from the heaviest shard (**select_load_to_move**) in order to **rebalance** chunks distribution with the new light shard (**select_target**).

Nasir *et al.*

Sources producing streams of data maintain in their *Load Information Registry* an estimate of the amount of load they sent to each worker. When a source produces a new message containing data, it triggers the process (*Imbalance Detector*) by applying two random hash functions on the message's key. Then, the source looks at its registry to retrieve (**get_load_report**) the amount of load it has recently sent to the two workers corresponding to these hash functions. The *Load Balancer* then selects the lightest worker (**select_target**) and sends it the message

(**select_load_to_move**) to be processed (**rebalance**).

AWS Auto Scaling

An Amazon CloudWatch sensor periodically sends a pull request (*Load Information Exchanger*) to the EC2 instances it monitors, in order to retrieve *load information* for a given set of criteria (CPU and RAM usage, latency, incoming network traffic, etc.). These metrics are collected into a repository (*Load Information Registry*). A CloudWatch alarm tracks the value of metrics and triggers an automatic scaling action whenever a threshold for a given metric is exceeded for a specific time duration (*Imbalance Detector*). The *Load Balancer* component will then be responsible for adding new EC2 instance(s) (**select_target**) that will receive part of the requests/traffic instead of the overloaded instance: for example, to decrease CPU utilization, part of the future requests received for the application will now be directed to the new instance (**select_load_to_move**). On the opposite, in case of an underload detected by CloudWatch (*Imbalance Detector*), the **select_target** method will pick an existing underloaded instance and terminate it.

5.4.2 P2P systems

Rao *et al.*

Peers periodically push their *load information* (*Load Information Exchanger*) to a set of nodes maintaining a directory (*Load Information Registry*). *Load information* contains the load of each virtual server of a peer and the peer's internal *threshold*. Each peer p periodically compares its load $load_p$ for a given load criteria to its $threshold_p$ (*Imbalance Detector*). Depending on the peer's load state, the paper proposes three rebalancing strategies (*Load Balancer*):

1. If $load_p < threshold_p$, p is underloaded and triggers a rebalancing. A random node is picked (**select_target**) and its load information is sent to p (*Pull Load Information* via *Load Information Exchanger*). If the random node is heavily loaded, then a virtual server transfer (**select_load_to_move**)

may take place between the two nodes (**rebalance**)[2].

2. If $load_p > threshold_p$, p is overloaded and contacts one of the peers holding a directory to request a light peer (**select_target**) and which virtual server (**select_load_to_move**) should be moved (*Pull Load Information* via *Load Information Exchanger*). Then, **rebalance** is called.

3. If $load_p > threshold_p$, p can also send its *load information* to a peer *dir* holding a directory (*Push Load Information* via *Load Information Exchanger*). After *dir* has received enough information from heavy and light nodes, *dir* performs an algorithm to pick (**select_load_to_move**) which virtual server p should send to which light node (**select_target**). The solution is then sent back to p, to start the **rebalance** process.

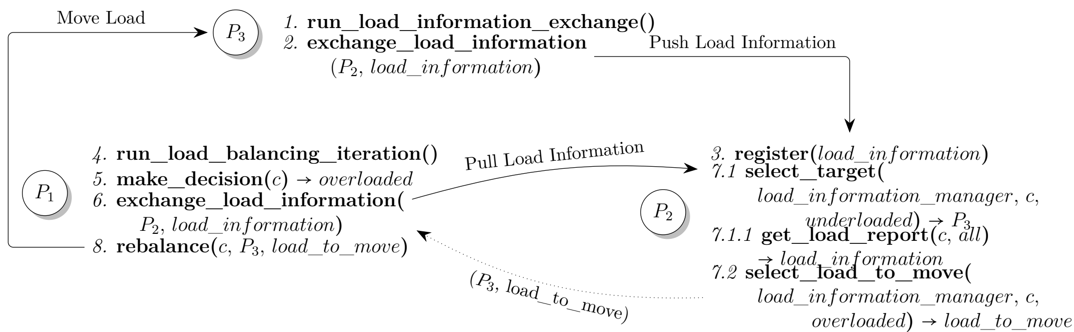


Figure 5.2 – Workflow associated with the second scheme proposed by Rao *et al.*

The workflow associated to the second strategy is depicted in Figure 5.2. Steps are numbered to sketch the sequence of actions involved in a typical load balancing iteration with three peers. Arrows between function calls depict remote communications. As a first step, the *Load Information Manager* component is used by $P3$ to send its load state to $P2$ which then stores the information in its directory (**register**, at step 3). Some time later, $P1$ evaluates its load state (**make_decision**) as *overloaded* (steps 4-5). In consequence, at step 6, $P1$ asks $P2$ for an underloaded peer which implies that $P1$ sends its load state to $P2$. Then, $P2$ looks for an underloaded peer that it knows of (this implies to call

[2]The same process can also occur if $load_p > threshold_p$ (in this case the heavy peer randomly looks for a light peer).

get_load_report at step 7.1.1), i.e. $P3$, and selects the amount of load that $P1$ should move to $P3$ so that both of them are well balanced. Finally, the result is sent back to $P1$ which can then contact $P3$ to start moving its load (**rebalance**). It is interesting to note the modularity offered by our API when implementing the scenarios proposed in this paper. Indeed, as all nodes do not play the same role (normal nodes or directory nodes), they do not all have to implement all components of the API. For instance, normal nodes (like P_1 and P_3 in Figure 5.2) do not need a Load Information Registry, thus it is only implemented on directory nodes (P_2).

Gupta *et al.* (Meghdoot)

Peers periodically exchange their load information with their neighbors (*Load Information Exchanger*), as well as an estimated list containing the most heavily loaded peers they know (from their *Load Information Registry*). Load balancing is only triggered (*Imbalance Detector*) when a new peer p wants to join the system. The first step of the rebalancing process (*Load Balancer*) is for p to find an overloaded peer in the system (**select_target**). To do so, p sends a *pull request* to a random peer. Then, the random peer will look at its registry (**get_load_report**) in order to tell p which node ($target$) is the most overloaded to its knowledge. Finally, $target$ is contacted by p. If the overload is due to the amount of subscriptions, p will split its zone with $target$ and receive half of $target$'s subscriptions. Otherwise, if $target$ is overloaded because of its processing load due to event propagation, p will replicate $target$'s zone and subscriptions (**select_load_to_move** and **rebalance**).

Byers *et al.*

A peer having to insert a data item into the system triggers the process (*Imbalance Detector*). This peer applies n hash functions on this item. Then, it contacts each peer associated to an hash function (*Load Information Exchanger*) to pull load information concerning the amount of items already stored by each of these peers. The *Load Balancer* then selects the lightest peer (**select_target**) and sends it the item to be inserted (**rebalance**).

5.5 Summary

In this chapter, we have described concepts behind the building of a generic API for load balancing in distributed systems. By decomposing a strategy into essential differentiators, we have shown how it is possible to abstract any behavior to fit our model. To present how the API could match existing load balancing strategies, we have used six different schemes from some representative solutions for Big Data storage and processing, based on P2P systems or not. Their strategies are triggered at various moments, consider different load criteria, require load information from various sources before taking a decision, and impact more or less nodes to rebalance the load. Although very different at first sight, most existing load balancing strategies rely on the same principles, regardless of whether they are implemented on a P2P overlay, a stream processing engine or even a Cloud infrastructure. We have shown it is possible to abstract and decompose any strategy into diffentiation criteria by answering three main questions:

1. *How is load information exchanged?* (differentiators (g) and (h)) requires to know who informs who (nodes) about what (load information), how (e.g., piggybacking, pull request) and when (periodically, or after some event).

2. *How to trigger load balancing?* (differentiators (a), (b) and (c)) gives information about what load criteria are taken into consideration (disk space, CPU usage, etc.), how is the load state estimated (comparison with an internal threshold or remote information) and when is triggered the decision to rebalance (periodically, or after new data is inserted, etc.).

3. *What has to be moved?* (differentiators (d), (e) and (f)) describes what and how much load has to be moved, but also which node(s) will receive this load.

Many different answers are possible for each of these questions, and hence an unlimited number of load balancing strategies are conceivable, simply by changing the behavior of one differentiation criteria. Accordingly, this leads to many possible implementations. Regarding the programming aspect, the API allows to separate the code concerning load balancing from the rest of the system. Thus,

implementing a different behavior for a strategy should be possible in a few lines of code. This feature is especially useful when developing a system for which one wants to try different load balancing strategies prior to a definitive choice, with the aim of finding the most efficient scheme. Also, as it is sometimes hard to predict the future performance of a system when designing, this makes it easy to adapt if any imbalance ever appeared later on.

An approach based on hierarchical components was presented, whose methods summarize the behavior of most existing load balancing strategies. This API is meant to be compatible with a wide range of distributed systems for data storage and retrieval. However, it is obvious that slight variations in the implementation of methods or the methods themselves might be necessary depending on the specifications of each system.

Chapter 6

Load Balancing Implementation and Experiments

Contents

6.1 Load Balancing Implementation on EventCloud . . . 100
6.1.1 Variations on components 100
6.1.2 Experiments 103
6.2 Experiments on Variable Hash Functions 108
6.2.1 Computing a new hash function 108
6.2.2 Load balancing policies 109
6.2.3 Experiments 114
6.3 Variable Hash Functions on a Chord Ring 125
6.3.1 Concept and specificities 125
6.3.2 Experiments 130
6.4 Summary 132

In this chapter, we present the various load balancing experiments we ran on different systems. All strategies were implemented using the generic API introduced in the previous chapter. First, we explain how we were able to easily add the load balancing functionality to the EC without affecting its existing business code. We also perform various experiments in order to find the most suitable strategy

to distribute the load among peers. Then, we consider the variable hash functions strategy presented in Chapter 4 and simulate some experiments to prove the efficiency of this solution in a CAN. We also show how performances can improve when modifying a single line of code, thanks to the genericity of the API. Finally, we perform similar experiments in a Chord overlay to show the same load balancing strategy (variable hash functions) can be implemented on a different overlay using nearly the same code.

6.1 Load Balancing Implementation on EventCloud

The work behind the generic API presented in the previous chapter was motivated by the building of our distributed platform for data storage and retrieval: the EC (introduced in 2.3). When developing this system, we found out load balancing had to be implemented to enhance the overall performance of the EC. More precisely, we observed that some of the benefits of our architecture (lexicographic order) also had their drawbacks, namely a poor data distribution among peers.
In this section, we assess the proposed API by extending the EC with the load balancing abstractions introduced previously. We give details about the load balancing strategies implemented in the EC and explain how the variation between each of them directly affects the behavior of three of the components presented earlier: the *Load Information Exchanger*, the *Imbalance Detector* and the *Load Balancer*. Then, some experiments are reviewed to gauge how flexible the generic API is.

6.1.1 Variations on components

The load balancing scheme we want to use in the EC consists in picking a new peer from a preallocated pool of peers to make it join an overloaded peer's zone in the network. For practical reasons, only one possible implementation of this strategy is presented in Algorithm 6.1. However, we detail below the other variants, too.

Load Information Exchanger

Load information exchange is an optional process. Two flavors are proposed: *absolute* and *relative*. With the *absolute* version, threshold values are configured per criterion and passed to peers when they are deployed and hence peers do not have to implement the *Load Information Exchanger* component. These thresholds are upper bound values that allow to signal an overload once they are exceeded. Concretely, defining such a behavior implies to set variables introduced in Algorithm 6.1 to specific values. By setting k to 1 and e to the desired threshold value (line 14), the **make_decision** function works with internal knowledge only. The *relative* version requires periodic load information exchange between neighboring peers to estimate the average load that each peer aims at remaining close to. Implementing this variant would require to switch line 14 with **Registry.get_load_report(c, local)**.

Imbalance Detector

This component is responsible for detecting whether a node is experiencing a load imbalance or not. We wanted to use a policy that allows to consider the unbalance of multiple different load criteria. Since **run_load_balancing_iteration()** is the entry point of the *Load Balancing Manager*, but also because imbalance detection is assumed to be performed periodically in the EC, the management of multiple load criteria is made at this level. The content of this function is summed up by Algorithm 6.1 along with its crucial subcalls. The entry point function detects (through the **make_decision** function) a peer as imbalanced if its load l for a criterion c is k times greater than a load estimate e associated to the criterion c that is observed. The retrieval of a meaningful e value is possible thanks to an internal threshold or information exchanged between peers (as presented in the previous paragraph). Variable k is a static variable scoped to the lifecycle of the system, like variable C which is a list of criteria defined before the system starts. The order in which the criteria are added matters since it defines priorities in which imbalances are detected. Besides, the detection process is sequential for the simple reason that load measurements are not necessarily expressed in the exact same unit but also the fact that the actions required to fix imbalances depend on

```
1:  function RUN_LOAD_BALANCING_ITERATION()
2:      for c ∈ C do
3:          load_state ← MAKE_DECISION(c)
4:          if load_state ≠ Normal then
5:              target ← SELECT_TARGET()
6:              load_to_move ← SELECT_LOAD_TO_MOVE()
7:              REBALANCE(c, target, load_to_move)
8:          end if
9:      end for
10: end function
11:
12: function MAKE_DECISION(c)
13:     l ← Registry.GET_LOAD_REPORT(c)
14:     e ← GET_INTERNAL_THRESHOLD(c)
15:     k ← GET_COEFFICIENT(c)
16:     if l ⩾ e × k then
17:         return Overloaded
18:     end if
19:     return Normal
20: end function
21:
22: function SELECT_TARGET()
23:     return Pool.GET_NEW_PEER()
24: end function
25:
26: function SELECT_LOAD_TO_MOVE()
27:     return GET_DATA_FROM_MIDDLE()
28: end function
```

Algorithm 6.1 – One of the possible implementations of the EventCloud load balancing algorithm relying upon the generic API. This implementation works as follows: each peer periodically runs the **run_load_balancing_iteration** function to compare its load for a set of criteria with an internal threshold. If the peer is overloaded, it asks a new peer (from a preallocated pool) to join its zone by splitting at the middle coordinate.

each criterion. Upon imbalance detection (line 4) for a single criterion at a time (in our case the first one relates to disk consumption due to the insertion of RDF data), the algorithm calls for functions from the *Load Balancer* component.

Load Balancer

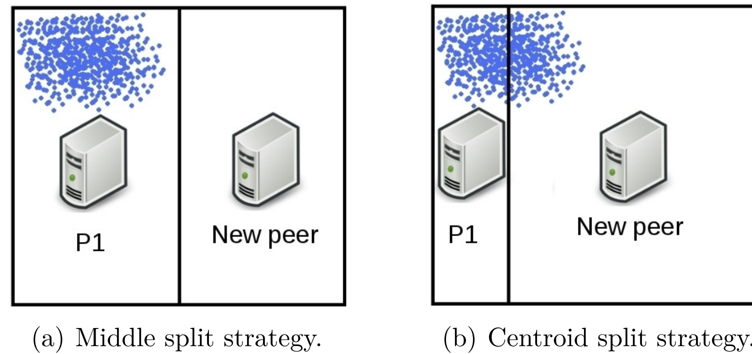
This component is responsible for moving some load from a node to another. In Algorithm 6.1, the **select_target** function is used to pick a peer from a pre-allocated pool of peers (line 5). Then, the load to move is selected with **select_load_to_move** (line 6). We implemented two different behaviors to partition the load between two peers: one that uses the middle value of the zone an overloaded peer is responsible for[1] and one that takes advantage of the centroid value, considering the number of items per peer and the RDF terms size (i.e. the number of characters). Figure 6.1 describes both methods, which may be seen as two different *Load Balancer* component implementations. These two variants apply to the **select_load_to_move** function description (lines 26-28). Algorithm 6.1 presents the variant implementing the middle split. However, when switching to the centroid split strategy, the **select_load_to_move** implementation would only require changing a single line of code (**get_data_from_centroid()** instead of line 27). Once the target and the load to move are identified, a **rebalance** is performed (line 7).

In summary, while remaining simple, the proposed load balancing strategy can lead to different implementations for the methods involved in the standard load balancing workflow and easily supports the definition of multiple independent criteria. For this reason, we propose to evaluate these variants in the next subsection to prove they can fit with the proposed API and lead to significant improvements in the distribution of data in the EC.

6.1.2 Experiments

Implementing our API on the EC has allowed us to try different load balancing policies during our experiments, corresponding to various implementations for

[1]This is the default CAN rule applied when adding a peer.



(a) Middle split strategy.   (b) Centroid split strategy.

Figure 6.1 – Middle vs. Centroid split strategies. Dots represent data items in the coordinate space managed by peers. In this example, we assume all data items are of a similar size. When using the middle split to add a new peer, peer P1's zone is divided into two zones of equal size: P1 will keep the left half of its zone and the new peer will become responsible for the other half. On the other hand, the centroid split is made according to the volume of data stored by P1, so that P1 and the new peer manage approximately the same amount of load.

some of the above-mentioned components. Variants of the proposed load balancing solution have been implemented and assessed with benchmarks using real data extracted from a Twitter data flow and up to 32 peers deployed on the French Grid'5000 testbed [70]. The workload is about 10^5 quadruples. The scheme consists in injecting the workload on a single peer and once all quadruples have been stored to start load balancing iterations.

Load Balancer variants

We ran a first experiment to evaluate various *Load Balancer* component behaviors. Each load balancing iteration is run by an oracle and consists in picking a new peer from a preallocated pool of peers to make it join the most loaded one in the network. The action is repeated until having a network containing 32 peers, which is still representative of existing applications. The number of peers was deliberately low as we first aimed to assess the usefulness of the API, not the load balancing strategies. To show the interest of using the centroid, the experiments have been performed by using zones cutting based on their middle or centroid values recorded on the fly. As shown in Figure 6.2, by cutting zones at their middle, the workload

is distributed on 4 peers only. However, the same experiment using centroid values distributes the load on all peers with almost two thirds having their load close to the ideal distribution. Although the distribution is not perfect, it is greatly improved. The main reason for such a difference between results for both strategies lies in the fact that RDF data is highly skewed. As depicted in Figure 6.1, when storing highly biased data like RDF terms using the lexicographic order, the distribution of data among the coordinate space is consequently highly biased as well. Although the middle split technique is the basic CAN load balancing operation and has already proven successful for some systems [54], in our case splitting a peer's zone in half according to the coordinate space does not improve the load distribution (Figure 6.1(a)). As shown in Figure 6.2, one of the four peers storing data still owns nearly all data items after 31 peers have been added. This is due to the fact that data covers a tiny area in this peer's zone and hence if it is located near a corner, many new peers will have to split with this node before actually starting moving data. A simplified version of this issue is depicted in Figure 6.3. On the contrary, when taking the specific context of our system into account by using the centroid split strategy (Figure 6.1(b)), data is necessarily distributed among peers.

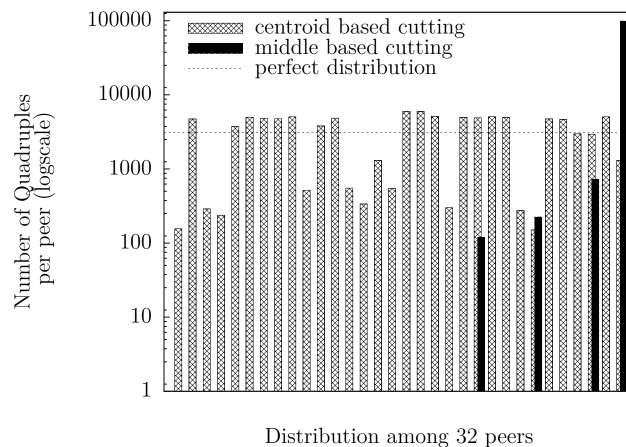
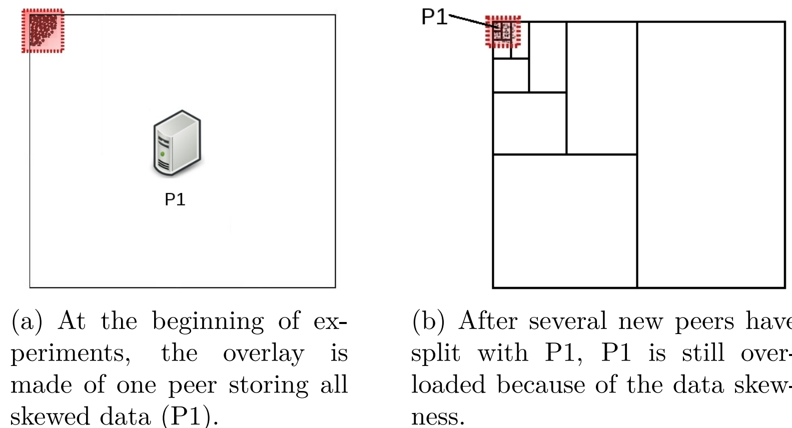


Figure 6.2 – Load balancing partitioning using middle vs centroid.

To compare results for a same configuration (i.e. same workload and number of peers), a good estimator is the coefficient of variation, also known as the relative standard deviation. It is expressed as a percentage by dividing the standard devia-

tion by the mean times 100. The smaller the percentage, the less there is disparity in terms of number of items stored by each peer. In the following, we use this estimator to compare strategies. As shown in Table 6.1, the coefficients are 559.4% and 69.5% when the middle and centroid methodologies are respectively applied to the load balancing experiments presented above. This clearly shows that the centroid performs better because its value is eight times lower than the middle value. Therefore, we used the *centroid* split strategy for our next experiment.



(a) At the beginning of experiments, the overlay is made of one peer storing all skewed data (P1).

(b) After several new peers have split with P1, P1 is still overloaded because of the data skewness.

Figure 6.3 – Middle split strategy unable to move data (the highlighted area) until many peers have been added.

Load Information Exchanger variants

We ran a second experiment based on a dynamic configuration to compare the two *Load Information Exchanger* variants. This time, each peer periodically estimates its load state using an internal threshold (*absolute* scheme) or load information received from its neighbors (*relative* scheme). Thus, peers do not rely on an oracle anymore to add the new peers. For the absolute strategy, the threshold value is set to the number of quadruples divided by the final number of peers, which gives 3125. For the relative strategy, k was set to 1.1, so that an overload is detected when local measurements on peers are greater than or equal to 1.1 times the estimate value computed by receiving load information from immediate neighbors. The parameter k was set according to empirical evaluations that let suppose the

best distribution is achieved for this value. For both strategies, an overloaded peer asks a new peer to join its zone by splitting at the centroid value.

	Oracle configuration		**Dynamic configuration**	
	Middle	*Centroid*	*Absolute*	*Relative*
Relative stddev	559.4%	69.5%	119.75%	96.57%

Table 6.1 – Load balancing strategies comparison in the EC.

Table 6.1 shows the results obtained according to the strategy applied, when correlated with the results obtained for the previous experiments that exhibit the best distribution that can be achieved (69.5%) due to the “oracle” assumption. The relative standard deviation is almost twice as large (119.75%) as the best when the absolute strategy is applied. Similarly, the relative strategy performs worse than the oracle-based load balancing solution but achieves a better distribution (96.57%) than the absolute strategy and this without using overall knowledge. These results are still satisfying and also more interesting because peers can rely on their own knowledge to make decisions.

Although the analysis of the results is not the central point of this section, it shows that investigating different implementations for the functions identified in Section 5.3 may have a strong impact on results. Thanks to the proposed API, the behavior of the different load balancing stages can be simply changed by writing a new method with less than 10 lines of code in our case. The main reason is that the key features of the load balancing workflow are clearly identified. Obviously, the examples shown in this section still require one line of code change and a full code recompilation to switch from a component implementation to another (and, consequently, the need to restart the distributed system from scratch). In our case, an alternative based on dynamic class loading could be used [71]. In this situation, some code redeployment is required but a whole system restart could be avoided. However, the synchronization between nodes has to be taken into consideration to prevent inconsistent states due to stale information that could transit during the transition from a component implementation to another on peers.

Although not measured during our experiments, we believe the possible overhead due to using the API when implementing load balancing remains negligible for its intended use in large infrastructures for Big Data storage and processing. Moreover, the aim of the API is rather to minimize the cost of development for experimenting various load balancing scenarios when compared to the cost of modifying a single large bloc of code.
Finally, two interesting perspectives are possible. The first is about the flexibility of the proposed API which can lead to implement adaptive load balancing strategies, for example, by changing one of the key features in a load balancing workflow at run-time [72]. The second is related to the properties of our system. Although it provides scale-up load balancing only, since events (RDF data) are stored in a sustainable manner, supporting scale-down load balancing (at the cost of an increase of the amount of data stored by each peer) by extending our solution would enable better resource usage through (even autonomic [73]) elasticity.

6.2 Experiments on Variable Hash Functions

In this section, we first explain how adaptive hash functions presented in Chapter 4 can be calculated to manage load imbalance, depending on various strategies. Throughout this section, we will consider a 3-dimensional CAN for RDF triples (in the form of *<subject,predicate,object>*) storage.

We propose variants of the load balancing strategy to implement with our approach. Then, we show how these variants are implemented using the generic API and how we can slightly vary their behavior. Finally, we present the experiments we ran to validate our approach. Thus, this section allows to validate in an experimental way both contributions of Chapters 4 and 5.

6.2.1 Computing a new hash function

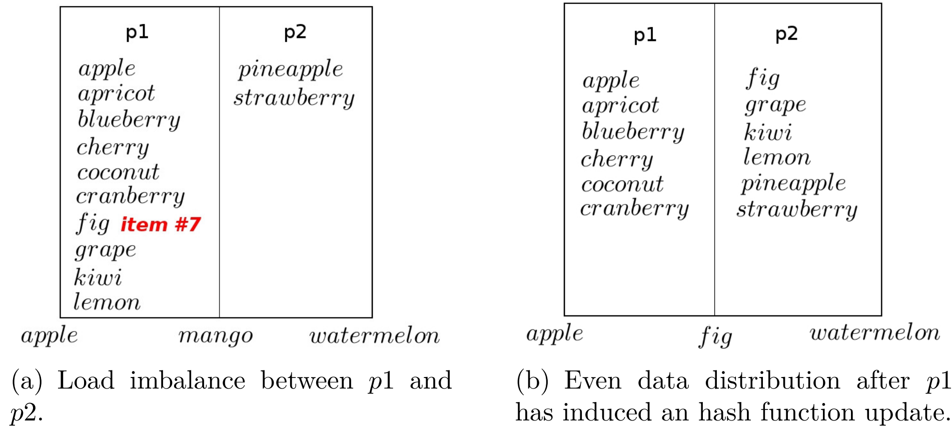
Following the principles set out in Chapter 4, we present in this subsection several possibilities for an overloaded peer to calculate the new hash function it wants to apply (i.e. the new value of its maximum Unicode bound). We consider the number of RDF triples a peer stores as the imbalance criterion, assuming all triples

are of a similar data size. A peer's load state will be either *normal* or *overloaded* if it stores too many RDF triples.
Let p be a peer storing a given number of triples noted as $load_p$ on a Unicode interval $[Umin_p, Umax_p[$. If p is overloaded, it has to send all triples above a certain limit, lim, to its neighbor(s) on the forward direction on a dimension[2] d, such as the new value of $load_p \leq lim$. Basically, p has to move its upper Unicode bound backwards, such as only lim data items remain in p's zone. This is achievable by simply changing p's hash function from $h1$ to $h2$ such that the new value of $Umax_p^d$ is equal to the Unicode value of p's $(lim+1)^{th}$ triple (as we consider p can sort triples alphabetically) and $h2(Umax_p^d) = max_p^d$. An example is shown in Figure 6.4, where we consider a peer is overloaded if it stores more than 6 triples. Thus, in Figure 6.4(a), peer $p1$ is overloaded and has to change its hash function on max_{p1}^d, corresponding to Unicode value *mango*. A new hash function $h2$ should transform $p1$'s 7th triple fig (as lim is equal to 6) into max_{p1}^d. In Figure 6.4(b), $p1$ and $p2$ have changed their hash function to $h2$ respectively on max_{p1}^d and min_{p2}^d, and their load is evenly balanced. However, it is important to note that Figure 6.4 depicts the best possible case where each peer ends up storing the same amount of items, which is much more complicated to achieve in practice, as we will explain later in this chapter.

6.2.2 Load balancing policies

We propose hereafter three different load information exchange strategies for the *Load Information Exchanger* component and their corresponding load state estimation behaviors a peer can use to check whether it should induce a rebalancing or not in its *Imbalance Detector* component. Then, we describe how the limit introduced above can be calculated for each scheme, corresponding to several possible implementations of the **select_load_to_move** function of the *Load Balancer* component.

[2] The dimension to reduce follows the same ordering as the default CAN ordering to handle node arrival: a peer first modifies its hash function on dimension 0, then if it is overloaded again dimension 1 will be reduced, and so on.



(a) Load imbalance between $p1$ and $p2$.

(b) Even data distribution after $p1$ has induced an hash function update.

Figure 6.4 – Load distribution before and after rebalancing. In this example, a peer is said to be overloaded if it stores more than 6 items.

Load Information Exchanger and Imbalance Detector

The proposed three different ways for peers to exchange load information directly affect, in our case, the way peers estimate their own load state in the **make_decision** function. Therefore, we present below these two aspects of a load balancing strategy together:

- **No information exchange between peers**
  Peers rely on an internal threshold to determine if they are overloaded or not. Thus, they do not need to implement the *Load Information Exchanger* component. A peer is said to be overloaded when it stores more RDF triples than the limit set by a threshold which has the same value for all peers (as we consider them all equal) within the network during our experiments. This *threshold* implementation of the *Imbalance Detector* component is presented in Algorithm 6.2.

- **Local exchange with neighbors**
  Local information exchange with immediate neighbors on a given dimension. Each peer p periodically contacts all its forward neighbors $Fneighbors_p^d$ on a dimension d to calculate the average data load $localAvg^d$ in its neighborhood for this dimension. If p finds out it stores more data than $localAvg^d$ and more data than a given threshold $localThres$, then p can induce a rebalance.

```
1: function MAKE_DECISION()
2:     l ← Registry.GET_LOAD_REPORT()
3:     t ← GET_INTERNAL_THRESHOLD()
4:     if l > t then
5:         return Overloaded
6:     end if
7:     return Normal
8: end function
```

Algorithm 6.2 – "Threshold" variant of the load state estimation algorithm when using variable hash functions.

```
1: function MAKE_DECISION()
2:     l ← Registry.GET_LOAD_REPORT()
3:     e ← Registry.GET_ESTIMATE(local)
4:     t ← GET_INTERNAL_THRESHOLD()
5:     if l > (t + e) then
6:         return Overloaded
7:     end if
8:     return Normal
9: end function
```

Algorithm 6.3 – "Local" variant of the load state estimation algorithm when using variable hash functions.

```
1: function MAKE_DECISION()
2:     l ← Registry.GET_LOAD_REPORT()
3:     e ← Registry.GET_ESTIMATE(overall)
4:     n ← GET_COEFFICIENT()
5:     if l ⩾ e × n then
6:         return Overloaded
7:     end if
8:     return Normal
9: end function
```

Algorithm 6.4 – "Overall" variant of the load state estimation algorithm when using variable hash functions.

We use $localThres$ to ensure a peer storing very few triples will not be allowed to change its hash function if it is surrounded by peers storing few triples, too. Indeed, we want to avoid unnecessary changes in the network whenever possible. However, as the data distribution and load are not known in advance, relying on a threshold may be problematic. One way to address this would be for peers to independently update their threshold value at runtime, depending on their knowledge of the system's overall or local load state. It is worth noting that $localAvg$ always refers to one dimension, so that p does not need to take into account the load of all its neighbors on all dimensions. This is due to the fact that a peer's hash function is modified on only one dimension at once, making it impossible for p to balance its load with all its neighbors at once. This *local* implementation of the *Imbalance Detector* component is presented in Algorithm 6.3.

- **Overall information exchange**
  In the present case, we assume each peer knows an estimate of the average network load, thanks to a gossip protocol or by asking a machine having an overall knowledge on the system. As the average overall load $overallAvg$ may be very low in a network made of thousands of nodes, we also use a coefficient n to consider a peer p is overloaded if it stores at least n times more triples than $overallAvg$. This *overall* implementation of the *Imbalance Detector* component is presented in Algorithm 6.4.

Load Balancer

When using variable hash functions, the way the **select_load_to_move** function is implemented in the *Load Balancer* component plays a crucial role as it is directly responsible for calculating the new hash function of a peer. Indeed, by deciding how many RDF triples it wants to move, an overloaded peer also implicitly determines its new hash function. To do so, we propose three different implementations:

- **Locally-based limit**
  A local rebalance means that a peer p will calculate how many triples $triplesToMove_p$

it has to send so that it roughly stores as many triples as $Fneighbors_p^d$. Then, max_p^d would be associated to the Unicode value of p's $(load_p - triplesToMove_p)^{th}$ triple.

- **Threshold-based limit**
  If a peer p is overloaded, it has to send all its triples above an internal threshold $thres$ to its neighbor(s) on a given dimension d. This approach is similar to the implementation on top of a skip graph proposed by [56]: max_p^d would be associated with the Unicode value of p's $(thres + 1)^{th}$ triple (i.e. p's first element above $thres$). This way, p would store $thres$ triples, and triples whose value on d is equal or above $Umax_p^d$ would be sent forwards in the network. For instance, in Figure 6.5, peer $p1$ is overloaded and has to get its data load under the specified threshold. To do so, $p1$ calculates it has to change its upper Unicode bound from M to K, such that $h2(''K'') = max_{p1}$. Then, all triples between K and M stored by $p1$ are moved to $p2$, whose data load will increase.
  During our experiments, $thres$ is the same value that determines both whether a peer is overloaded (load state) and how much load has to be moved (corresponding to the limit value introduced in 6.2.1). However, two different values of $thres$ could be used, too.

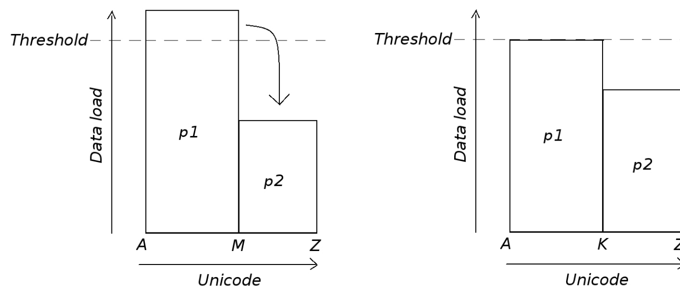


Figure 6.5 – Threshold-based limit strategy.

The threshold-based approach is more geared towards storage systems undergoing periodic bulk inserts, as it may trigger a new hash function calculation quite often in the case of continuous data insertion. In order to avoid repetitive rebalancing, alternatives based on a value lower than $thres$ can be used to determine,

and thus to increase, the amount of data to move. Such value can correspond for instance to the median or the centroid value of the data stored by the concerned peer.

- **Median-based limit**
  In the present case, the overloaded peer p uses the Unicode value of its median triple as its new value for $Umax_p^d$. This way, half of p's triples would be sent forwards in the network, without underloading p. Alternatives using the average or the centroid value of a peer's data items could also work similarly.

To summarize, infinite possibilities may be considered to calculate the amount of load to move. For our experiments, we will focus on the three different behaviors previously introduced. They correspond to three possible implementations of the **select_load_to_move** function and each one is presented in Algorithm 6.5. To show the influence of this method on the choice of a peer's new hash function, we describe a simplified version of the whole hash function update algorithm in the **run_load_balancing_iteration** method. If a peer estimates it is overloaded (**make_decision**), it first selects the amount of load to move using one of the three presented behaviors. The first item of this list (alphabetically sorted) corresponds to the new upper Unicode bound of the overloaded peer on the chosen dimension to reduce $Umax^{dim}$. Then, the peer starts routing the information about this update (**propagate_new_hash_function**) over the concerned CAN bound on this dimension max^{dim}. Finally (lines 8-11), the peer attempts to forward each item in *load_to_move* to the right neighbor as soon as its neighbor(s) on max^{dim} use the correct hash function (this is not carried out immediately as it may take some time before they update).

6.2.3 Experiments

Experimental set-up

To validate our approach, we ran extensive experiments in a cycle-based simulator: PeerSim [74]. A cycle represents the time required by a peer to perform a basic operation (message routing, load checking, etc.). We simulate a 3-dimensional

```
1:  function RUN_LOAD_BALANCING_ITERATION()
2:      load_state ← MAKE_DECISION()
3:      if load_state ≠ normal then
4:          dim ← GET_NEXT_DIMENSION_TO_REDUCE()
5:          load_to_move ← SELECT_LOAD_TO_MOVE(dim)
6:          Umax^dim ← load_to_move[0]
7:          PROPAGATE_NEW_HASH_FUNCTION(dim, max^dim, Umax^dim)
8:          for each item ∈ load_to_move do
9:              target ← SELECT_TARGET(item)
10:             REBALANCE(target, item)
11:         end for each
12:     end if
13: end function
```

Threshold-based limit implementation

```
14: function SELECT_LOAD_TO_MOVE(dim)
15:     return GET_DATA_FROM_THRESHOLD(dim)
16: end function
```

Locally-based limit implementation

```
17: function SELECT_LOAD_TO_MOVE(dim)
18:     local_avg ← Registry.GET_LOAD_REPORT(local, dim)
19:     even_avg_load ← COMPUTE_EVEN_AVG_LOAD(local_avg)
20:     return GET_DATA_FROM(even_avg_load)
21: end function
```

Median-based limit implementation

```
22: function SELECT_LOAD_TO_MOVE(dim)
23:     return GET_DATA_FROM_MEDIAN(dim)
24: end function
```

Algorithm 6.5 – Hash function update algorithm.

CAN composed of 1000 peers. It is iteratively built by starting with a single peer, and new peers always split with the largest available zone. This ensures that the network is as balanced as possible, i.e. all zones are of similar sizes. Although being modelled on the EventCloud architecture, our system is implemented on a simulator in order to perform large-scale experiments with many peers concurrently modifying their hash function at the same time.

Our CAN uses the lexicographic order to disseminate data: the maximum Unicode value supported U_{max} is set to character number 2^{20} on all dimensions, to encompass the whole Unicode characters table. In order to make realistic experiments, we used a dataset made of triples from three different sources. The first two are only composed of Latin Unicode characters, from the Berlin BSBM benchmark [75] and the LC Linked Data Service[3]. The third source is extracted from DBpedia [8] and contains Japanese Unicode characters. One million triples are inserted into the network in the space of 15 cycles, to simulate bursty traffic with a continuous insertion of data. As load balancing should occur periodically, the **run_load_balancing_iteration** function of peers was called every 5 cycles, in order to allow time for overloaded peers to send part of their data before evaluating their load state again.

A perfect data distribution would correspond to all 1000 peers storing 1000 triples. However, this is not achievable in practice for two reasons. First, the CAN topology may not be perfect (some zones larger than others). Secondly, real RDF data is often very skewed, due to the large scope of the whole Unicode and the fact that RDF triples may contain very similar values, like ID values that differ by a single character. For instance, many triples in the Berlin benchmark contain values like $ProductFeature1$, $ProductFeature2$, which can go up to $ProductFeature4745$ in the data file we use. This makes balancing even harder, especially considering the specificity of RDF triples, whose *predicate* values are often not very distinct. For instance, thousands of different triples could share the same *predicate*: $typeOf$, specifying the RDF type of the subject. Overall, our datasets contained less than 50 different values of *predicate*.

[3] `http://id.loc.gov/`

Originally, we ran experiments for each of the three load information exchange strategies described in 6.2.2, to first validate our approach. For the threshold-based approach, we set a threshold of 8000 triples and also used this value as the limit of load to move (threshold-based limit). For the locally-based approach, we set a threshold of 30000 triples and applied the locally-based limit. For the overall-based strategy, we used a coefficient of 15, which means a peer was overloaded if it stored more than $overallAvg \times 15$ triples, and used the median-based limit to move triples. Threshold/coefficient values were set according to empirical evaluations that let suppose a satisfying distribution is achieved for these values. Lower values could be used but would require more time and communications to obtain results. Conversely, higher values could be chosen in order to achieve balance sooner. Depending on the chosen load information exchange strategy, peers periodically multicast (to their neighbors) or broadcast their load state (i.e. the number of items they store).

Reference algorithm

In order to show the efficiency of our solution, we also ran an experiment to compare our results with those obtained when using a commonly used load balancing strategy relying on a unique and linear hash function. The scheme consists in adding a new peer at the location of an overloaded peer. This technique is the default CAN load balancing strategy and is used by famous storage systems such as Meghdoot [54]. We chose this strategy because it is commonly used by CAN overlays. Also, it is natively compatible with the architecture of our simulated system and the EC, unlike node or data replication solutions that would require some adjustments due to the use of the lexicographic order on all dimensions.
To perform experiments using this reference algorithm, we consider an overlay composed of a single peer, at the beginning. Then, periodically (every 5 cycles in our case), PeerSim adds a new peer to the system at the most loaded peer's location. To find this peer, we simulated an oracle providing this information, as this is not the central point of our experiments. When a new peer arrives, the middle split strategy (previously introduced in Figure 6.1) is used and data that does not

belong to the overloaded peer anymore is moved to the new peer. We performed our experiments by starting with one peer receiving the one million triples, then new peers were periodically added until having a network made of 1000 peers. For this strategy, there is no particular balance to achieve as peers do not compare their load with a threshold. Load balancing is triggered by an external event (a new peer joining), thus experiments were stopped once all peers were added.

Results

Strategy	**Peers storing data**	**Changes of hash function**	**Moved triples**	**Cycles to achieve balance**	**Standard deviation (triples)**
None	2	0	0	n/a	235702
Threshold	652	156	10333076	80	1618
Local	818	170	5153092	100	2675
Overall	602	90	4626023	45	1900
Add peers	407	0	9334233	n/a	1487

Table 6.2 – Data dissemination over peers and cost of each load balancing strategy to achieve balance.

Table 6.2 summarizes results obtained for each load balancing strategy at the end of simulations. Results are expressed in terms of number of peers storing data (out of 1000), hash functions updates sent by overloaded peers[4], cumulated total number of triples moved from peers in order to balance their load, cycles to achieve balance[5] (after all triples are stored by the very first peer) and standard deviation in terms of RDF triples stored.

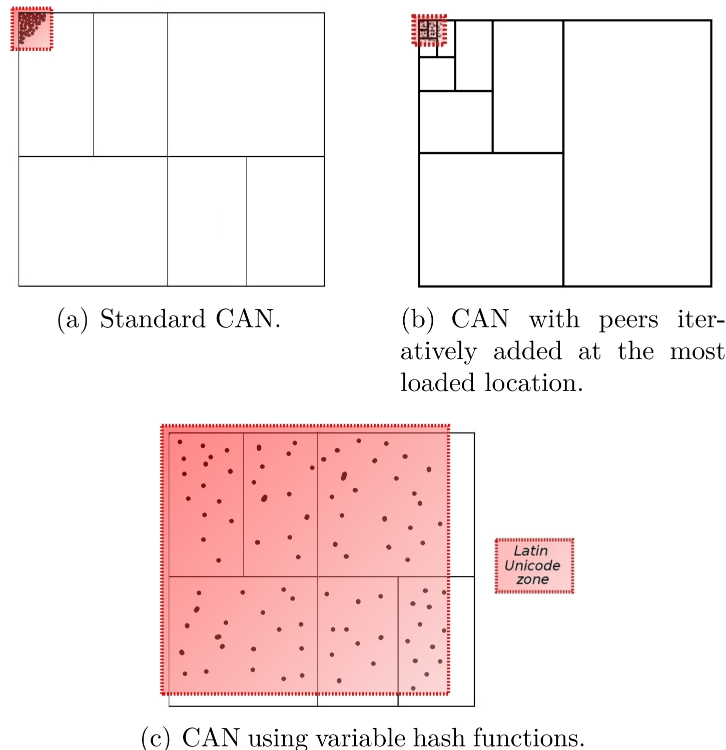
Without applying any load balancing strategy, results show that only 2 peers would store data, because of the large bias between datasets (Latin vs. Japanese

[4]This value does not necessarily represent the number of updates uniformly applied in the overlay: some updates may be rejected by peers while routing, due to a concurrent update.

[5]By achieving balance, we mean no more peer is overloaded, according to a given strategy.

characters). These two peers respectively correspond to the one at the far-left of the CAN on each dimension, responsible for storing all Latin triples, and the peer at the far-right of the CAN on each dimension, responsible for storing all Japanese triples.

When using variable hash functions, the threshold-based strategy distributes the load over 652 peers and offers the lowest standard deviation between the load of peers, but requires moving a large number of triples (over 10 million). The locally-based strategy distributes data across more peers than the other strategies (818), but has a higher standard deviation and requires slightly more updates on hash functions (170). As a consequence, it takes more time to achieve balance. The overall-based strategy distributes the load over 602 peers, thus requires less communication between nodes (only 90 changes of hash function and 4626023 triples moved). This also means peers will store more data, but balance will be achieved sooner.



(a) Standard CAN.

(b) CAN with peers iteratively added at the most loaded location.

(c) CAN using variable hash functions.

Figure 6.6 – Load distribution depending on the strategy used.

When adding peers at the most loaded locations, i.e. when using a linear hash function shared by all peers, data is far less well distributed (407 peers). The main reason for this lies in the fact that the skewed data is mapped to an extremely small area in the CAN, as shown in Figure 6.6(b), depicting a simplified version of the CAN after several peers have joined the most loaded peer. This brings us back to the skewed distribution issue known by standard CAN overlays presented in Figure 6.6(a) and also earlier in Chapter 4. Even though many peers are added to this area, this solution still remains less efficient than when using variable hash functions (Figure 6.6(c)). By adding new peers, data is moved from one peer to another but the items' CAN coordinates still remain the same as the hash function applied by all peers on data does not change. This considerably limits data dissemination and, at some point, becomes ineffective.

Using variable hash functions required to move an important volume of data to balance the load. However, our results show that a more conventional strategy fails to achieve better results in terms of data items moved between peers: around 9 million triples were moved when adding peers against approximately 4.5 and 5 million for the local and overall schemes. Therefore, our solution, although not perfect, is not more costly regarding data transfer than other existing load balancing strategies. Moreover, it has the advantage that it does not require data to be moved far away in the overlay because only one hop is necessary to move data to a direct neighbor. To date, generally speaking, it remains impossible to address the load imbalance issue regarding data storage without having to move large amounts of data.
Regarding the communication cost related to the periodic load information exchange between peers (except when using an internal threshold), our solution requires peers to send information either to their neighbors only or to all peers. When using the *add peers* strategy, although we simulated an oracle for our experiments, a similar process to ours would be needed in a real system (like Meghdoot) to help a new peer find an overloaded zone. Thus, when a peer joins the system, it usually contacts an existing peer that must have some knowledge about the load of other peers in the network. This necessarily implies periodic load information exchange between peers with the *add peers* strategy as well. Consequently, both

strategies (variable hash functions or new peers added) should approximately require the same amount of communications throughout the network.

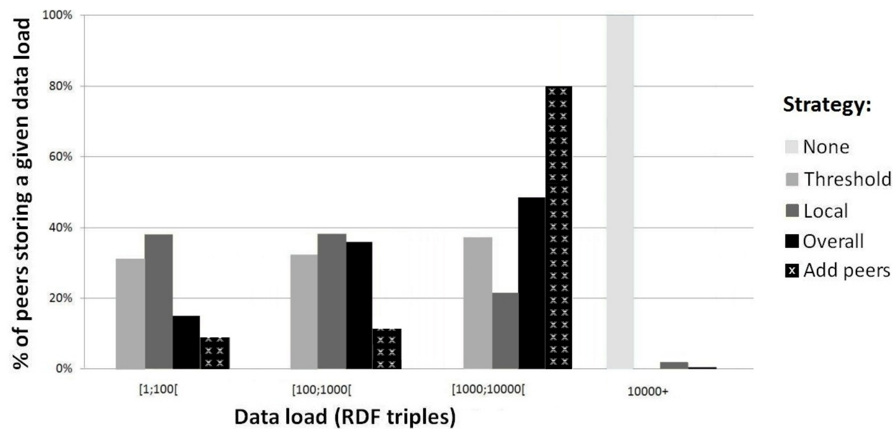


Figure 6.7 – Load distribution (excluding peers storing no data)

Figure 6.7 represents the load distribution at the end of our experiments, excluding peers storing no data. We divided the number of triples stored by each peer into 4 intervals ($[1;100[, [100;1\,000[; [1\,000;10\,000[; [10\,000;1\,000\,000]$) and reported the percentage of peers storing the corresponding amount of data. For example, using the threshold-based strategy, 31% of peers storing data hold between 1 and 99 triples, 32% store between 100 and 999 triples, 37% between 1000 and 9999, while no peer stores more than 9999 triples. The aim of Figure 6.7 is to show the efficiency of our solution for disseminating data using variable hash functions, with almost no peer storing more than 1% of the total amount of data stored in the network at the end of experiments, for all strategies.

However, in practice, such a wide dissemination might not be necessary for some reasons. Indeed, such dissemination requires to move a lot of data items throughout the overlay, which brings us to the cost issues of our solution, especially regarding communication, data transfer and overall system performance. In a real system, moving large amounts of data to achieve a perfect balance state would take some time and might impact the performance of the system while rebalancing, regarding bandwidth or processing capacities. Nonetheless, this issue is experienced with any load balancing strategy as it is inevitably necessary to move load when

trying to lessen the load of a peer. Finding a good tradeoff between the cost of moving data and the cost of having overloaded peer(s) in the network is challenging and mostly depends on which aspect has the strongest effects on the system performance. Moreover, we try to show in our experiments what could be the best possible distribution, while it is often necessary to offload only a small number of peers to improve performance, which means this would imply less communication and data transfer than in our experiments. By doing so, fewer communications between peers would be necessary to solve range queries for instance, rather than when having a wide range of values disseminated over a wide range of peers. Once again, it is a matter of finding the optimal, or at least the right degree of dissemination for a particular system. This problem raises the more general question of oscillation when trying to balance the load. In our case, this effect would not be exclusively caused by the load balancing scheme being used but also by the whole environment around the system like the periodicity of data insertion, the threshold/coefficient being used, or even the number of peers and their respective storage, bandwidth and processing capacities. Furthermore, load balancing might not have to happen with great frequency, depending on how in the existing code is triggered the **run_load_balancing_iteration** function call. In our case, this method was called periodically but it would have been possible to call it only after the insertion of a large volume of data on a peer, for example. The oscillation effect is outside the scope of this thesis and will not be further discussed later on.

To ensure that the load balancing operations do not affect the consistency of the overlay, 200 random queries were sent to random peers throughout the experiments. For all of them, the correct result was received within a reasonable time (an average of 15 hops to route queries or results, hence 15 cycles). The number of update bound messages created by peers and the fact these updates may not be applied by all peers at the same time did not affect the resolution of queries within our system. This is because a triple is always stored at least on one peer. Moreover, moving data is only done once an update bound message is applied by both the sender and the receiver. Thus, reaching the new responsible peer for a given triple should not require more than a few hops in the network, at the very most.

None of our strategies offers a perfect load balance which would correspond to all 1000 peers storing 1000 triples. Alternatives to these problems could be to remove the *predicate* dimension in our CAN, or even to reduce the Unicode scope of the network (U_{max}) and redirect non-matching triples (i.e. triples outside this reduced interval) to specific peers. For example, in a CAN entirely dedicated to the Latin Unicode interval, a peer could be responsible for storing all data made of non-Latin characters in case of rare non-Latin data insertion.

Generic API benefits

To show the interest of implementing a load balancing strategy using our generic API, we performed, in a second phase, experiments to determine how to improve the results of Table 6.2. We switched the various behaviors regarding how the load information is exchanged, how a peer estimates its load state and how an overloaded peer selects its load to move. Previously, in 6.2.2, the ways to evaluate the load state and the amount of load to move were both directly linked to the *Load Information Manager* component implementation, i.e. the way load information was exchanged (or not). This time, it is interesting to note that a peer may rely on a particular source of load information to estimate its load state (neighbors, all peers or none), while using another source to select the load to move.

For our experiments, based on the same context as 6.2.3, we focused on the number of peers storing data at the end of simulations, as this represents one of the most significant values to evaluate the efficiency of a load balancing strategy. However, the same principles could be applied to measure other parameters, such as the amount of transferred data or the elapsed time to achieve balance. Such information may also be very useful in order to choose the most suitable strategy for a particular system.

Table 6.3 presents the different results regarding data distribution among peers, depending on the two behaviors chosen for estimating the load state of peers and selecting the load to move. Results show that the data distribution can be notably

Load to Move \ Load State Estimation	Threshold	Local	Overall
Threshold	652	433	508
Local	715	818	945
Median	672	577	602

Table 6.3 – Variation regarding the data distribution when using various implementations based on the generic API. This table shows the number of peers (out of 1000) storing data at the end of experiments depending on how a peer evaluates its load state, associated to how an overloaded peer selects the amount of load to move.

improved depending on these two behaviors. Therefore, by modifying a minor parameter at first glance (e.g., is the load to move selected depending on a threshold or the median value), important variations in terms of distribution may appear between strategies. Regarding the programming aspect, switching from one strategy to another is trivial, thanks to the genericity of our API. In our case, usually only a few lines of code had to be modified to do so (see Algorithms 6.2, 6.3, 6.4 and 6.5).

As regards the results of these experiments, it is interesting to note that a better distribution occurs when taking into account the load information of other peers to calculate the amount of load to move. When being associated to an overall knowledge of the network load to estimate a peer's load state, the distribution is almost perfect, with 945 peers storing data out of 1000. As such knowledge might not be achievable in practice, relying only on the load information received from neighbors or even on no external information to estimate a peer's load state also seems to offer a convincing data dissemination (respectively 818 and 715 peers concerned). This means we could consider a strategy with no load information exchange before a peer gets oveloaded and, only in this case, messages would then be exchanged with neighbors to estimate their load and hence, the amount of load to move. Such strategy could be regarded as efficient for a system whose peers only know a very small subset of the network (their neighbors), and have few or no contact with other peers. Indeed, with this strategy, no communication between

peers is needed to exchange load information as long as no peer gets overloaded. Moreover, the load information exchange would be restricted as an overloaded peer would only contact its neighbors to determine the best amount of load to move.
In contrast, when using external information to determine a peer's load state but relying on an internal threshold (median or a set limit), the benefits of having external knowledge are lost when comes the time to rebalance the load. Consequently, these configurations offer the lowest dissemination: 433 peers when estimating the load state according to the neighbors load and the load to move according to a threshold. When relying only on internal information to determine the load to move, it seems more efficient to also rely on internal information to estimate a peer's load state. However, these results could vary in a system where peers have different capacities, and hence different thresholds.
Many other variants are possible depending on what are the key issues for a particular system: conversely, it is conceivable to opt for a variant offering a lower dissemination in order to offload some machines while avoiding large data transfers.

6.3 Variable Hash Functions on a Chord Ring

So far, we have presented our approach using several hash functions on a CAN overlay and how to implement it with different behaviors using a generic API. In this section, we will show it is also possible to use the API for implementing the same load balancing solution (variable hash functions) on a different overlay (Chord).

6.3.1 Concept and specificities

Presentation

Let us consider a Chord [18] ring for Unicode-encoded data storage. More precisely, we will consider a system for RDF data storage, to keep in line with the work presented so far. Each node n, represented by a fixed *Node Id*, is associated with a *key* identifier, and should store all items whose hashed value is comprised between $]key_{n.predecessor}; key_n]$. The default hash function used in Chord destroys the natural ordering of data that we want to preserve. Thus, our default hash

function will maintain the lexicographic order. On the ring, this means that nodes are assigned key identifiers ordered according to the lexicographic order. Consequently, it becomes impossible to guarantee an efficient skewed data dissemination, as we introduced earlier in the context of CAN, hence the need to use variable hash functions. Other Chord-based solutions to deal with this issue while using a non-randomizing hash function include virtual rings [53] or the use of a *leave-join* mechanism [55].

In our Chord, the key refers to an attribute of the data being stored, hence this technique may be applied more generally for any NoSQL data storage for instance. In our example, the key corresponds to the value of an RDF triple's object. We chose the *objects* as their values are not as biased as *predicates*, hence this should result in a better distribution. A triple t will be located at $successor(t_{object})$: the successor peer of t's object value. As keys correspond to only one part of a triple, routing a query or a triple can be made according to this part only. We opted for this simple architecture because our main goal is to show our variable hash functions strategy can be compatible with other overlays. However, other alternatives could be considered to determine how data should be stored, such as having several virtual rings [76] for each part of a triple, or indexing a triple three times (i.e. each time indexed using a different part of a triple), as proposed by [41].

Variable hash functions

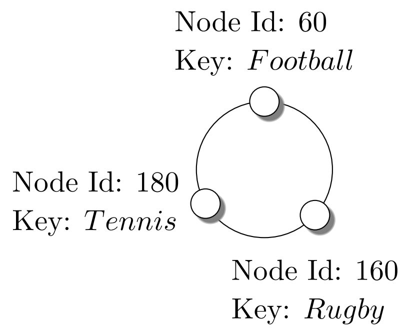
Previously, we introduced our notion of hash function in a CAN overlay, corresponding to the mapping between Unicode values and CAN coordinates. Using variable hash functions means that a given CAN coordinate, which is fixed, is associated to a Unicode value that may change over time. As there is no notion comparable to a fixed CAN interval in a Chord overlay, we use the node identifier (*Node Id*) as the fixed value to be associated with the node's key identifier, which can vary. Unlike the default Chord implementation that assigns node and key identifiers using a consistent hash function, we separate the assignment of node identifiers from the assignment of key identifiers. By default, the key identifier

of a node is computed using its node identifier and, very often, the ambiguous notion of *identifier* is used when referring to either a node or key identifier. In our implementation, both terms refer to two different notions. A *node identifier* refers to a node and its value can be computed by applying a consistent hash function on the node's IP address (we consider this value should not change), like in any Chord overlay. The *key identifier* of a node refers to the maximum key value managed by this peer. We allow a peer's key identifier to change over time by using variable hash functions. This implies, among other things, that peers should dissociate node and key identifiers when routing a message: in our overlay, routing should only be done according to a peer's key identifier. Therefore, the knowledge maintained by peers about their neighbors or finger table entries should include both node and key identifiers. More precisely, a finger table entry in our Chord implementation should contain the node identifier and the IP address of a peer (as in any Chord overlay) to identify a node, but also the key identifier associated to this peer in order to resolve lookups.

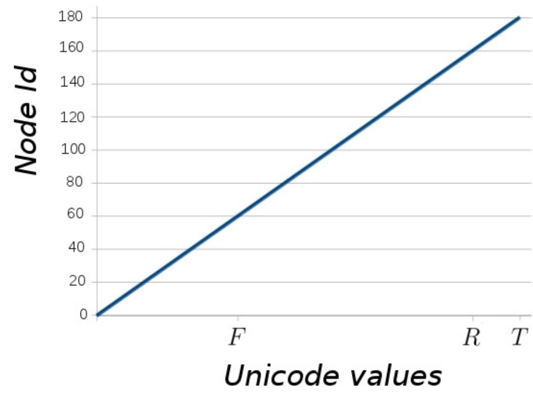
Whenever a peer is overloaded, according to a given load balancing strategy (the same as those described in 6.2.2), this peer can reduce its key identifier value in order to store less data. For example, in Figure 6.8, $Node$ 160 ($N160$) has reduced its key identifier to store only triples whose object value is between interval $]Football; Golf]$. As a consequence, its successor $Node$ 180 has a wider Unicode interval to manage ($]Golf; Tennis]$).

Propagation and stabilization

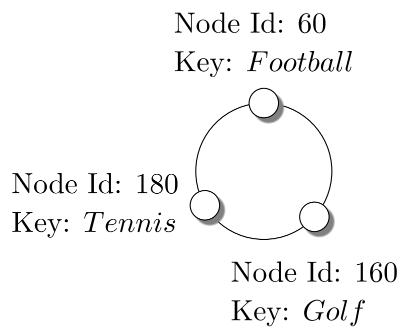
As presented in 4.3.6, a peer updating its hash function must notify its neighbors. On a Chord ring, this means $N160$ has to notify all its successor(s). As there is no notion of dimension in Chord, there is no need to propagate this information any farther to maintain the overlay consistent. The new hash function, depicted in Figure 6.8(d), requires $N180$ to capture part of $N160$'s keys (hence also data items) between $]Golf; Rugby]$. Unlike a propagated update in a d-dimensional CAN that may require to move data on the whole modified dimension, a hash function change in a Chord overlay only requires data movement between a peer and its successor.



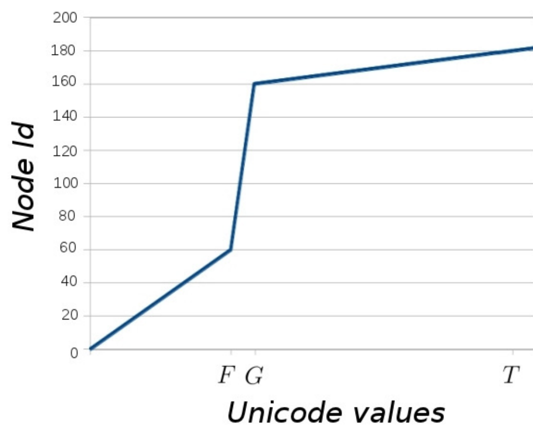
(a) Default Chord.



(b) Default hash function.



(c) Chord after *Node* 160 has reduced its key from *Rugby* to *Golf*.



(d) New hash function.

Figure 6.8 – Variable hash functions on a Chord overlay.

However, other peers might still include $N160$ in their finger table associated with key value *Rugby*. As the finger table link is unidirectional, $N160$ does not know which peers are concerned, thus cannot directly inform them of its new key identifier (i.e. its new hash function). A similar problem is experienced by the default Chord overlay when a new peer joins the system and captures some keys but existing nodes' finger table entries are not up-to-date. To address this issue, Chord uses a stabilization protocol to help peers maintain correct information about their successor, predecessor and finger table entries. By default, each node periodically runs this protocol to learn about new nodes. In our case, they also learn about new hash functions (i.e. new key identifiers). The stabilization scheme is composed of two main functions:

- *stabilize*() consists for a peer in checking whether what it knows about its successor (node, and hence key identifier) is correct, otherwise the peer updates its knowledge. In Figure 6.8(c), this means $N60$ will have to update its information about its successor $N160$'s key now equal to $Golf$.

- A $fix_fingers()$ method proceeds similarly to refresh finger table entries. In our case, this method also updates the key identifier associated with a peer in the finger table, if necessary.

As stabilization occurs periodically, it may take some time before a key update propagates to all the concerned finger table entries. Nevertheless, in the meantime, this has minor impact on lookup performance. If a query for item $Handball$ is received by $N160$ from a peer that does not know about the new hash function yet, $N160$ will forward the query to its successor $N180$ which is responsible for $Handball$. In this simple case, routing will require one more hop. In a more complex configuration, that is to say if x successors of $N160$ also reduce their key identifier before all fingers on $N160$ are adjusted, routing may require up to x more hops. In this case, the best recommendation to maintain adequate lookup performance is to repeatedly verify fingers correctness, as suggested by the original Chord paper.

Torus topology

Regarding the last peer in the Chord ring (the one with the highest *Node Id*), its default key identifier should correspond to the maximum Unicode character supported U_{max}. Nonetheless, this peer is also allowed to reduce its key identifier thanks to a process similar to the one presented for CAN in Section 4.3.2. Let Pn be the last peer in a ring made of n peers. Thus, Pn manages values up to U_{max}, that corresponds to its key identifier key_{Pn}. Let $P1$ be the first peer in the ring order, managing values from U_{min} to its key identifier key_{P1}. If Pn gets overloaded, it can reduce its key identifier and $P1$ will then be responsible for two intervals of keys (just like the CAN approach), for keys comprised in $]key_{Pn}; U_{max}]$ or $[U_{min}; key_{P1}]$. On the ring, this has little impact as keys are still ordered according to the lexicographic order. More generally, this should not affect the network consistency as long as a peer p's predecessor has a key identifier lower than key_p or p is responsible for the U_{min} key.

6.3.2 Experiments

Experimental set-up

Our experiments aim at showing it is also possible to implement the variable hash functions strategy to balance the load of a Chord overlay. The same context of the CAN experiments in Section 6.2.3 was used: the same datasets were inserted in a network made of 1000 peers. The original Chord paper recommends that each node maintains a list of successors to increase robustness in case of a node failure, in our case we used three successors per node. At the beginning of experiments, the default key identifiers are evenly assigned to peers by following the lexicographic order, such as all peers are responsible for approximately the same number of keys between U_{min} and U_{max}. Then, each peer would periodically compare its load state using either the internal, local or overall estimation scheme. The internal threshold was set to 1000 items. The local scheme would consider a peer as overloaded if it stored more than 1300 items and also more items than the average load of its three successors. Finally, the overall scheme would trigger load balancing if a peer stored more than twice the average network load. We were able to use lower

threshold/coefficient values than for the CAN experiments as the distribution is based on the object values of triples only, which are not as biased as the very few distinct values of predicates, for instance.

Results

Strategy	**Peers storing data**	**Changes of hash function**	**Moved triples**	**Standard deviation (triples)**
None	2	0	0	235702
Threshold	1000	1328	23589693	3743
Local	806	1874	23098537	6353
Overall	760	1339	21313525	1141

Table 6.4 – Load balancing results for each strategy applied on a Chord ring.

Table 6.4 summarizes results when considering keys based on the object value of triples. The threshold strategy allows to share workload among all peers as we consider only one dimension based on the object of triples, whose value is much more distinct than the predicate value. Such perfect distribution was obtained as the volume of data to be inserted was known beforehand, which helped us estimate the perfect threshold. The goal of such experiment was to show how efficient variable hash functions can be to disseminate data in a Chord overlay. However, in practice, it might be harder to distribute data over 100% of the nodes if the amount of data to store is not known in advance.

The local strategy, which measures the load of a peer's successors to decide whether to induce a rebalance or not, was able to distribute data among 806 peers. The overall strategy offers the lowest standard deviation between the 760 peers storing data at the end of experiments. These two schemes were less effective to distribute data as too low threshold/coefficient values would have caused an infinite oscillation, hence the need to use higher values that may not trigger load balancing as often as expected to obtain a perfect distribution.

For all strategies, the number of triples to move and the number of hash functions

changes were higher than those obtained when using a CAN overlay, for several reasons. First, the better distribution among peers consequently involves more triples to be moved between peers. More importantly, triples are moved from successor to successor, as they have to apply the new hash function, which may lead them to rebalance their load as well. Finger tables, that would help save some hops, are not updated as quickly and hence are not as efficient as usually. Finally, in a CAN overlay, a hash function update is propagated on the whole dimension, which means many peers may potentially benefit from this change. Indirectly, a single overloaded peer may prevent many other peers from being overloaded when asking them to modify their hash function, and hence these peers will not need to initiate their own update later on. In our Chord approach, such saving is impossible and each overloaded peer has to trigger its own hash function update.

To conclude, we have shown it is possible to implement, thanks to the generic API, the same load balancing strategy presented in Algorithm 6.5 on a different overlay (Chord). To do so, no major modification is needed: *neighbors* become *successors* and the propagation is done towards successors only.
These experiments allowed us to evaluate if the variable hash functions strategy is as efficient in Chord as in CAN. Results have shown it appears more costly in Chord than in CAN as concerns the amount of load to move and the number of hash function updates induced by peers. From another point of view, we tried to distribute the load as much as possible while it may not be necessary to try and evenly balance the whole network (offloading some peers might suffice). Moreover, existing load balancing alternatives based on peer duplication or relocation are not cost-free either. Thanks to the genericity of the API, one may implement these other schemes in order to evaluate which one offers the best tradeoff between efficiency and cost, for a particular system.

6.4 Summary

Throughout this section, we have presented various experiments performed to evaluate various load balancing strategies on P2P-based storage systems. All schemes were implemented using our generic API introduced in Chapter 5. Using this

generic API, an unlimited number of different load balancing strategies may be implemented, by slightly modifying a strategy. This change can concern the way load information is exchanged, when is load balancing triggered, how much load is moved, who receives the load to move, and so on. Our goal was to improve data distribution, which is meant to increase the number of nodes involved to answer queries/subscriptions and hence also improve performance towards end-users.

The work behind the API was motivated during the development of our own distributed system for Semantic Web data storage and retrieval: the EC. Due to the natural skew of RDF data and the specificities of the EC architecture, we found out load balancing had to be implemented on the EC. To do so, we did not want to affect the existing business code while being able to try on various strategies in order to choose the most efficient one to disseminate data among peers. We have shown in this chapter that components behaviour can easily vary by implementing a function differently in the API. On the EC, we mainly focused on two aspects of a load balancing strategy for which we varied the implementation: how is the load to move selected and do peers exchange load information to help them determine their load state. Significant differences were observed in terms of distribution among peers, especially regarding the ways to select the load to move.

In the second section of this chapter, we presented how to implement the variable hash functions strategy introduced in Chapter 4 in a CAN overlay using the generic API. We presented various possible implementations to calculate a new hash function and estimate a peer's load state. Some of them require external knowledge about the load of peers in the overlay while others only rely on internal thresholds. We performed experiments in order to validate our approach and compare it with an existing famous load balancing scheme consisting in adding a new peer at the location of an overloaded peer. Results have shown variable hash functions can distribute data over twice as many peers as this famous strategy, and represented a 400-fold increase over when using no load balancing strategy. Regarding the cost of our scheme, it does not require more communication between peers or data transfer than existing strategies to rebalance storage load and, unlike some of them, has the benefit of not changing the overlay topology (i.e. we

maintain the same neighboring links).
Then, we performed another set of experiments to show how to easily improve such performances when varying some components behavior. By changing a couple of lines of code, we were able to double the number of peers storing data in some cases. Interestingly, this shows how essential are some parameters that at first sight may seem of absolutely minor importance.

Finally, we implemented the same variable hash functions strategy in a Chord overlay. The purpose of this was to show the same scheme, and hence the same code, could be deployed in a different overlay thanks to the genericity of the API. Even though each overlay/system has its own specificities, we believe the separation of concerns offered by the API can also ease the deployment of an existing strategy onto another system. Results in Chord have shown that data can also be well distributed but at a higher cost regarding the amount of data transfer and the time to achieve balance, as many more hash function updates had to be initiated than in CAN. Although we tried to distribute data as much as possible while it may only be necessary to offload some machines in a real system, these results also suggest that the same load balancing strategy might not have the same efficiency on different systems. Therefore, it should be useful to be able to try and experiment various load balancing strategies in a particular system in order to choose the most satisfying one, which is the main purpose of our API.

Chapter 7

Conclusion

Contents

7.1 Summary 135

7.2 Perspectives 137

7.1 Summary

This thesis proposed contributions around load balancing in structured overlays. This work was motivated during the development in our research group of the EventCloud, a distributed platform for Semantic Web data storage and retrieval. We have presented a new solution based on the use of multiple order-preserving hash functions to improve the dissemination of skewed values. To easily implement and modify it, as well as any other load balancing scheme, a generic API was also presented.

Variable hash functions

When storing highly skewed values like Semantic Web data in an order-preserving manner, the distribution of data among peers is consequently highly skewed as well. Since hash functions are at the heart of data distribution in DHT-based overlays, we proposed to address this issue by allowing an overloaded peer to modify the

hash function it applies on data, so that it can reduce the interval of values it is responsible for. This results in the coexistence of several different hash functions used by different peers. In order to maintain optimal routing and consistency in the overlay, we have shown that it is not necessary for all peers to use the same hash function as long as some rules are observed. To our knowledge, this is the first load balancing solution for structured overlays based on the coexistence of multiple order-preserving hash functions to improve data dissemination. To evaluate the effectiveness of this solution, we simulated an overlay composed of 1000 peers and injected 1 000 000 highly skewed RDF triples. Results have shown that the number of peers storing data is 2 out of 1000 when using no load balancing strategy. When implementing the variable hash functions solution, up to 945 peers store data at the end of experiments. We also compared our solution with a well-known strategy consisting in adding a peer at the location of an overloaded node. Our solution appeared more efficient to disseminate the load and also less costly regarding the amount of data to move to achieve balance.

Generic API

Implementing a load balancing strategy in a distributed system is not as simple as one might think. There exists a wide range of load balancing solutions. However, each system has its own architecture and also its own performance requirements regarding various criteria. Thus, implementing an existing strategy may not give the desired results as each system is different and hence better solutions may be implemented and evaluated, for instance by fine-tuning a strategy until the performance requirements are met. To do so, we presented in this thesis a generic API for implementing load balancing in distributed systems. The API has been built after the analysis of existing solutions which led us to identify differentiation criteria like *how is the load state of nodes evaluated?* or *how is the load to move selected?*. These criteria represent the right level of abstraction to model any strategy and led us to determine some components exposing a set of well-defined functions. Modifying a strategy usually consists in modifying the behavior of one or few differentiation criteria. When using our API, this simply consists in re-

implementing the functions corresponding to these differentiators. Therefore, this does not affect the existing business code and, from a development point of view, eases the possible changes when compared to modifying a hard-coded strategy. The experiments we performed to implement, among others, the variable hash functions strategy have shown that results can be significantly improved depending on the behavior of some components. Moreover, switching from one behavior to another is trivial as it usually consists in re-implementing a function of the API, which required in our case to modify only a few lines of code.

7.2 Perspectives

We performed our experiments on both CAN and Chord overlays but we believe that the use of variable hash functions could be applied to other DHT-based overlays. Likewise, we only used RDF data in our experiments but the same principles could be followed with any other Unicode-encoded data representation (e.g., key-value pairs, tuples). We presented this strategy in the context of a system placing data in the overlay according to the lexicographic order. Other types of sorting could be considered like the ascending or descending order for numerical values. Also, the bound(s) associated with each peer may not be based on the value of data items but on their format or size. Thus, it might be possible to use this technique for non Unicode-encoded data, as long as the hash function is able to return a coordinate/identifier to locate the item in the overlay.
We have shown variable hash functions can help improve data dissemination. Concerning some possible future experiments, it might be interesting to evaluate with some use cases the benefits of this improved data dissemination on the processing load of peers. Also, we considered as our main load criterion the number of items stored by a peer to estimate its load state but another criterion could be considered: the popularity of the items stored by each peer. Indeed, the popularity of some particular sets of items being frequently accessed may generate processing load for the peer storing them. An alternative to replication would be to allow a peer storing few items but overloaded because these are popular items to modify its hash function so that part of them are sent to its neighbor(s) that may store many items but with low access frequency.

Finally, throughout this thesis, we mainly considered load balancing for *overloaded* peers. However, the variable hash functions strategy could be extended to *underloaded* peers as well. The scheme would consist in allowing an underloaded peer to modify its hash function in order to enlarge the interval of values it is responsible for.

Currently, we envision to explore the use of variable hash functions in the context of stream processing. Our research team works on the building of a distributed architecture for analyzing RDF streams. The resulting architecture could look like what is presented in Figure 7.1. This system considers streams of RDF data produced by various sources like Twitter flows or sensor networks. The goal of the system is to evaluate in real time whether this incoming data matches some queries (e.g., subscriptions). To achieve this faster, a query is divided into subqueries taking the form of a triple so that subqueries can be executed in parallel and their respective results are then merged to answer the initial query. Each subquery is associated with a *worker* node responsible for matching the incoming triples against this subquery. To efficiently process the streams and the queries, each of these nodes is exclusively responsible for triples and subqueries containing a defined value of predicate. The data streams would be randomly received by the machines in Layer 1, then sent to dispatcher nodes in Layer 2 that could be organized into a structured overlay like Chord or a skip list. Each dispatcher would be responsible for a given interval of predicate values and hence would receive all incoming triples matching this interval (in Figure 7.1, the letter written into each node represents the node's upper bound). Finally, these dispatchers would send to workers (Layer 3) the triples containing the predicate value they are responsible for in order to start the subquery matching process to eventually merge the results.

We are exploring the possibility of implementing a load balancing scheme on this future system to prevent some workers or dispatchers from becoming overloaded by streams or even by too many subqueries with the same popular predicate. Since one of the main advantages of using variable hash functions is to help balance the load among nodes to adapt in real time to incoming data, this strategy may be considered as a load balancing solution in this context. For example, if a

node at Layer 2 receives too many triples because it manages the most popular interval of predicate values, it may modify its hash function to receive less triples and hence reduce its processing load (as data is not meant to be stored). Alternatively, other load balancing solutions could be experimented using our generic API, such as those proposed by Nasir *et al.* [47] (power of two choices paradigm to pick the least loaded node to receive the stream), Meghdoot [54] (adding nodes at the locations being overloaded by subscriptions) or even a brand new solution designed according to the principles proposed with the generic API.

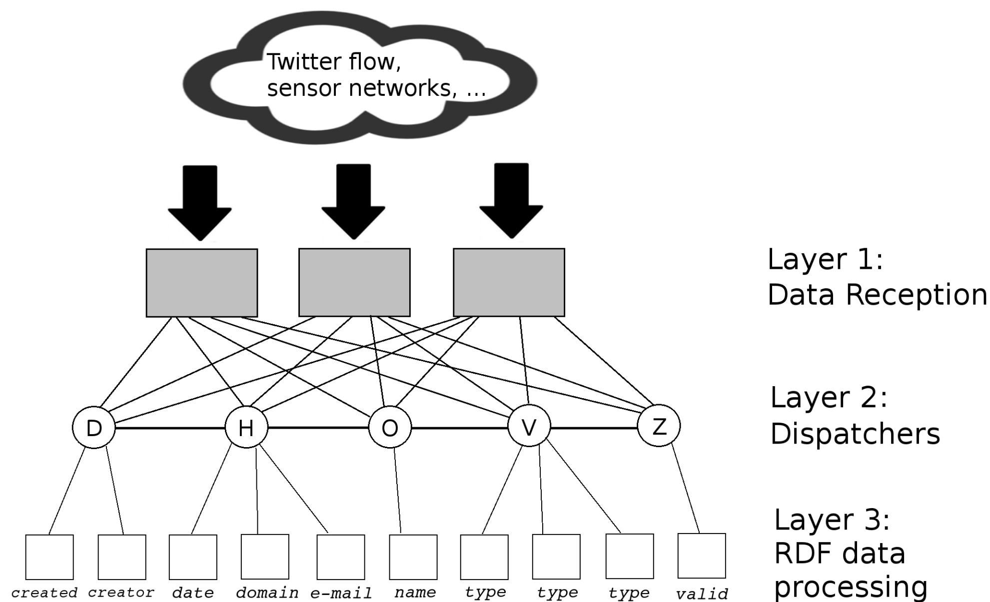


Figure 7.1 – Possible RDF stream processing architecture.

Annexe A

Version française

Amélioration de la Dissémination de Données Biaisées dans les Réseaux Structurés

Sommaire

A.1 Introduction 142

A.1.1 Motivation et Problématique 142

A.1.2 Plan et Contributions 144

A.2 Résumé du contenu de la thèse 146

A.2.1 Fonctions de Hachage Variables 146

A.2.2 API Générique pour l'Equilibrage de Charge 147

A.2.3 Implémentation et Expériences 149

A.3 Conclusion 151

A.3.1 Résumé 151

A.3.2 Perspectives 153

A.1 Introduction

A.1.1 Motivation et Problématique

De nombreuses applications ont besoin d'intégrer des données du Web pour en extraire de l'information et des connaissances. L'utilisation de données du Web peut se rapporter aux réseaux sociaux, au partage de contenu multimédia ou encore en intelligence d'affaires. Parmi les nouvelles technologies du Web 3.0, le Web Sémantique apporte une signification à chaque élément du Web et propose des outils utiles à la représentation des connaissances et au raisonnement sur les données du Web. Comme l'Internet grandit de manière exponentielle et que de plus en plus de données sont générées chaque jour, la notion de Big Data a vu le jour pour faire référence à de larges collections de données hétérogènes produites par diverses sources. Par conséquent, ces données sont généralement faiblement structurées, d'où la possibilité que les différences de taille, popularité ou contenu entre les ressources puissent varier de manière importante. Pour cette raison, les jeux de données du monde réel, y compris ceux produits sur le Web, sont connus pour potentiellement contenir des données fortement biaisées.

Avec l'avènement du Big Data, il devient très difficile de gérer des volumes de données réalistes sur une seule machine. Pour faire face à cet incroyable volume d'information et en tirer parti, tout en offrant des performances suffisantes aux utilisateurs dans des contextes différents, de nombreuses solutions distribuées existent, dont les systèmes Pair-à-Pair (P2P), les bases de données NoSQL distribuées, les services dans les nuages ou les moteurs de traitement de flux. Tous ces systèmes représentent des solutions efficaces à large échelle pour le stockage et le traitement de données dans des environnements distribués. Le travail derrière cette thèse a été largement influencé par le developpement dans notre équipe de recherche d'un système P2P pour le stockage et la récupération de données du Web Sémantique, orienté vers l'adaptabilité situationnelle sous la forme d'une plateforme de publication d'évènements.

Cette thèse traite de la question liée à la dissémination de données biaisées entre les nœuds, qui est un problème de plus en plus fréquent dans les systèmes distribués. Avec l'émergence du Web Sémantique et du Big Data, les données sont souvent stockées d'une manière qui préserve leur ordre afin de permettre la récupération de valeurs proches sur des nœuds voisins, ce qui limite le nombre de sauts nécessaires à l'exécution d'une requête. En conséquence, le même sous-ensemble de nœuds est systématiquement contacté pour les données/requêtes entrantes, ce qui peut surcharger ces nœuds. En se basant sur les constats observés sur notre plateforme distribuée, qui utilise une telle architecture et doit faire face à ce problème de déséquilibre de charge, nous proposons dans cette thèse une nouvelle approche pour améliorer la distribution des données parmi les nœuds d'un Réseau Structuré (en anglais : SON pour Structured Overlay Network). Notre technique préserve l'ordre des données et ne requiert pas de déplacer de nœud, ni de répliquer des données, qui sont des solutions communes mais coûteuses en ce qui concerne la mise à jour de la topologie du réseau ou le maintien de la cohérence des données. Notre contribution vise à gérer dynamiquement les déséquilibres de charge en permettant à différents pairs d'utiliser différentes fonctions de hachage, tout en maintenant le réseau consistant. Dans les SONs, la fonction de hachage appliquée par les pairs pour insérer ou récupérer un objet est au cœur de la distribution de données. Cependant, peu de solutions d'équilibrage de charge reposent sur l'utilisation de plusieurs fonctions de hachage, et habituellement celles-ci visent à répliquer des données populaires. Dans notre approche, nous autorisons un pair à changer sa fonction de hachage dans le but de réduire sa charge. Puisque cela peut être réalisé pendant l'exécution, et sans connaître au préalable la distribution naturelle des données, cela représente un mécanisme d'équilibrage de charge efficace et adaptif.

L'équilibrage de charge est un facteur clé pour n'importe quel système distribué, en particulier ceux orientés vers la dissémination de données. Les déséquilibres sont souvent causés par une répartition inégale des clés/identifiants réseau, l'arrivée et le départ fréquent de nœuds, ou encore l'hétérogénéité entre ces derniers en termes de bande passante, de capacité de stockage ou CPU. Comme les données produites

ont un besoin grandissant d'être analysées/manipulées/récupérées en temps réel par un nombre potentiellement très élevé d'utilisateurs, s'attaquer au problème du déséquilibre de charge est nécessaire, surtout pour minimiser les temps de réponse et éviter que le travail ne soit réalisé que par seulement un ou quelques nœuds. Pour ce faire, il existe presque autant de stratégies d'équilibrage de charge qu'il y a de systèmes différents. De plus, les solutions proposées sont souvent couplées à leur propre API, ce qui rend difficile le portage d'une stratégie d'un système à un autre. Pourtant, lorsque l'on développe un système distribué, nous pensons qu'il serait utile de pouvoir essayer différentes solutions, car il n'est pas toujours aisé d'anticiper quelle stratégie serait la plus efficace et adaptée à un système particulier. Dans cette thèse, nous proposons de définir une API commune pour implémenter n'importe quelle stratégie d'équilibrage de charge indépendamment du reste du code. Nous montrons que beaucoup de stratégies sont constituées des mêmes éléments, et que seules l'implémentation et l'interconnexion de ces éléments varient. En définissant correctement ces éléments et leur comportement, un nombre illimité de possibles stratégies d'équilibrage de charge pourrait être conçu et implémenté.

A.1.2 Plan et Contributions

Cette thèse traite de l'équilibrage de charge dans les systèmes distribués pour le stockage et le traitement de données. Sa contribution est principalement organisée autour de trois chapitres, qui sont les chapitres 4, 5 and 6. Nous présentons d'abord notre stratégie d'équilibrage de charge basée sur des fonctions de hachage variables pour la dissémination de données biaisées. Ensuite, nous proposons une API pour facilement implémenter n'importe quelle stratégie d'équilibrage de charge dans un système distribué. Enfin, nous détaillons les expériences que nous avons effectuées avec cette API, en particulier pour implémenter notre stratégie des fonctions de hachage variables. Globalement, cette thèse est organisée comme suit :

- Le **Chapitre 2** donne une vue d'ensemble des concepts évoqués dans cette thèse. Nous présentons d'abord la notion de Web Sémantique, en nous focalisant sur les moyens de représenter et requêter les données. Ensuite, nous présentons le Pair-à-Pair et ses différents types de réseaux. Enfin, nous décri-

vons l'architecture de l'EventCloud, un système Pair-à-Pair pour le stockage et la récupération de données du Web Sémantique développé dans notre équipe de recherche.

- Le **Chapitre 3** présente un état de l'art de solutions pour équilibrer la charge dans les systèmes distribués orientés Big Data. Nous décrivons les principales techniques existantes pour contrer les déséquilibres touchant à la distribution des tâches parmi les nœuds. Nous nous focalisons surtout sur les systèmes P2P pour le stockage et la récupération de données biaisées/populaires et/ou préservant l'ordre naturel des données, pour rester en corrélation avec l'architecture et les problèmes connus par l'EventCloud.

- Le **Chapitre 4** décrit la solution que nous proposons contre les déséquilibres de charge concernant la distribution de données biaisées dans les systèmes P2P préservant l'ordre naturel des données. Comme les fonctions de hachage sont au cœur de la distribution des données dans les systèmes basés sur une DHT (Table de Hachage Distribuée), nous proposons d'utiliser différentes fonctions de hachage en même temps pour améliorer la dissémination des données biaisées. Notre but est de permettre à un pair surchargé de modifier la fonction de hachage qu'il applique sur les données, afin qu'il réduise l'intervalle de valeurs dont il est responsable et donc qu'il ait moins d'objets à stocker. Cette contribution a été présentée à la conférence PDCAT en 2014 [1].

- Le **Chapitre 5** présente une API générique pour implémenter la plupart des stratégies d'équilibrage de charge dans les systèmes distribués pour le stockage de données. Nous montrons comment n'importe quelle stratégie peut être décomposée en critères et comment le changement de comportement de ces critères permet de créer un nombre illimité de stratégies différentes. L'API est utile pour passer d'une stratégie à une autre en quelques lignes de code, ce qui peut être utile lorsque l'on développe un système, pour facilement expérimenter différents comportements. Ce travail a tout d'abord été présenté dans le cadre des ateliers SBAC-PAD 2014 [2]. Puis, une version étendue a été publiée dans un numéro spécial du journal édité par Wiley

Concurrency and Computation : Practice and Experience en 2015 [3].

- Le **Chapitre 6** présente les expériences que nous avons effectuées avec notre API générique. Nous montrons comment l'équilibrage de charge a été implémenté sur l'EventCloud, ainsi que les différentes stratégies que nous avons testées avant de finalement choisir la plus efficace pour distribuer des données du Web Sémantique entre les pairs. Puis, nous présentons nos expériences simulées dans des réseaux CAN et Chord pour améliorer la distribution des données en utilisant des fonctions de hachage variables.

- Le **Chapitre 7** conclut cette thèse. Il passe en revue les contributions présentées et ouvre la perspective vers de nouvelles initiatives de recherche qui pourraient résulter de ce travail.

A.2 Résumé du contenu de la thèse

Le travail abordé dans ce manuscrit de thèse est principalement développé dans trois chapitres. Ci-après est présenté un résumé de chacun de ces chapitres.

A.2.1 Fonctions de Hachage Variables

Dans le Chapitre 4, nous présentons notre stratégie d'équilibrage de charge visant plus particulièrement à améliorer la distribution de données biaisées parmi les nœuds. Les fonctions de hachage sont au cœur de la distribution des données dans les systèmes basés sur une DHT. Cependant, peu de stratégies d'équilibrage de charge s'attaquent précisemment à la question des fonctions de hachage dans l'idée d'améliorer la dissémination des données. De plus, ces stratégies visent à répliquer des données ou détruisent l'ordre naturel des données. Elles ne visent pas directement le problème des données biaisées alors que les données du monde réel sont connues pour être fortement biaisées [67] [10] [68]. Notre approche a pour but de disséminer des données biaisées sans connaître à l'avance la distribution naturelle de ces données.

Dans la plupart des systèmes P2P, les pairs n'ont pas une connaissance globale du réseau. Nous avons montré qu'il n'est pas nécessaire que tous les pairs utilisent la même fonction de hachage, tout en continuant de maintenir le routage optimal et le système cohérent. Les pairs n'ont même pas besoin de connaître la fonction de hachage globale utilisée dans le réseau. Pour ce faire, quelques règles sont nécessaires pour s'assurer que les pairs reçoivent suffisamment d'information sur ces changements tout en restant capables de prendre les bonnes décisions sans aucune communication préalable avec d'autres pairs.

Lorsqu'un pair est surchargé, nous lui permettons de réduire l'intervalle dont il est responsable en modifiant la fonction de hachage qu'il applique sur les données, afin qu'il puisse en envoyer une partie vers son ou ses voisins. Entre temps, un algorithme de multicast optimal présenté en Section 4.3.3 est utilisé pour propager cette nouvelle fonction de hachage seulement vers les pairs concernés par cette mise à jour (Section 4.3.1). Les destinataires prennent indépendamment la décision d'accepter ou de rejeter cette mise à jour. Cependant, ils finissent tous par prendre la même décision, même dans le cas de plusieurs mises à jour concurrentes (Section 4.3.4). Grâce à ces règles, nous pouvons facilement limiter les communications entre pairs tout en maintenant le système consistant. Chaque pair applique sa propre fonction de hachage lors du routage d'une requête pour une donnée. Ainsi, localiser la position d'un objet se fait dynamiquement, à la volée : chaque pair routant le message applique sa propre fonction de hachage, ce qui règle le problème du manque de connaissance des pairs concernant la fonction de hachage globale du système.

A.2.2 API Générique pour l'Equilibrage de Charge

Dans le Chapitre 5, nous décrivons les concepts derrière la création d'une API générique pour l'équilibrage de charge dans les systèmes distribués. En décomposant une stratégie en critères de différenciation, nous avons montré qu'il est possible d'abstraire n'importe quel comportement pour se conformer à notre modèle. Afin de présenter comment l'API pourrait être compatible avec des solutions d'équilibrage de charge existantes, nous avons utilisé six stratégies différentes représenta-

tives de celles utilisées par des systèmes pour le Big Data, basés sur du P2P ou non. Ces stratégies sont déclenchées à divers moments, prennent en comptes différents critères de charge, nécessitent de l'information provenant de diverses sources avant de prendre la moindre décision, et impactent plus ou moins de nœuds lors du rééquilibrage. Bien que très différentes à première vue, la plupart des stratégies existantes reposent sur les mêmes principes, peu importe qu'elles soient implémentées sur un système P2P, un moteur de traitement de flux ou même une infrastructure de type Cloud. Nous avons montré qu'il est possible d'abstraire et de décomposer n'importe quelle stratégie en critères de différenciation en répondant à trois questions essentielles :

1. *Comment est échangée l'information sur la charge ?* (critères de différenciation (g) et (h)) permet de savoir qui informe qui (nœuds) sur quoi (information sur la charge), comment (par exemple avec du piggybacking ou des requêtes envoyées à des nœuds) et quand (périodiquement, ou après un certain évènement).

2. *Comment déclencher l'équilibrage de charge ?* (critères de différenciation (a), (b) et (c)) permet de savoir quel(s) critère(s) de charge est pris en considération (espace disque, utilisation CPU, etc.), comment est estimé l'état de charge (comparaison avec une limite interne fixée ou avec de l'information récupérée à distance) et quand est déclenchée la décision de rééquilibrer (périodiquement, après que de nouvelles données aient été insérées, etc.).

3. *Quelle est la charge à déplacer ?* (critères de différenciation (d), (e) et (f)) décrit la charge à déplacer, mais aussi quel(s) nœud(s) recevra cette charge.

Beaucoup de réponses différentes sont envisageables pour chacune de ces questions, et donc un nombre illimité de stratégies d'équilibrage de charge sont concevables, simplement en changeant le comportement d'un critère de différenciation. Par conséquent, de nombreuses implémentations sont possibles. En ce qui concerne l'aspect programmation, l'API permet de séparer le code concernant l'équilibrage de charge du reste du système. Ainsi, implémenter un comportement différent pour une stratégie doit pouvoir se faire en quelques lignes de code. Cet aspect est utile

par exemple lorsque l'on développe un système pour lequel on veut essayer différentes stratégies d'équilibrage de charge avant de faire un choix définitif, dans le but de trouver la solution la plus efficace. Aussi, comme il est parfois difficile de prévoir à l'avance les futures performances d'un système, cela permet de s'adapter plus facilement si des déséquilibres devaient apparaître par la suite.

Une approche basée sur des composants hiérarchiques a été présentée, dont les méthodes résument le comportement de la plupart des stratégies existantes. Cette API a été conçue pour être compatible avec un large éventail de systèmes distribués pour le stockage et la récupération de données. Cependant, il est évident que de légères variations dans l'implémentation des méthodes ou les méthodes elles-mêmes pourraient être nécessaires en fonction des spécificités de chaque système.

A.2.3 Implémentation et Expériences

A travers le Chapitre 6, nous avons présenté diverses expériences effectuées pour évaluer differentes stratégies d'équilibrage de charge sur des systèmes de stockage P2P. Chacune d'entre elles a été implémentée avec notre API générique présentée dans le Chapitre 5. Grâce à cette API, un nombre illimité de stratégies d'équilibrage de charge différentes peut être implémenté, en appliquant de légères modifications à chaque fois. Ces changements peuvent concerner la façon d'échanger de l'information sur la charge, quand est déclenché l'équilibrage de charge, combien de charge doit-on déplacer, qui reçoit cette charge, et ainsi de suite. Nous recherchons à améliorer la distribution des données, ce qui est censé faire augmenter le nombre de nœuds impliqués dans les réponses aux requêtes/souscriptions, et donc agit sur les performances vers les utilisateurs finaux.

Le travail derrière cette API a été motivé durant le développement de notre propre système de stockage distribué pour des données du Web Sémantique : l'EventCloud (EC). A cause du fait que les données RDF sont naturellement biaisées et aussi dû à l'architecture spécifique à l'EventCloud, il fut nécessaire d'implémenter un mécanisme d'équilibrage de charge sur l'EC. Pour ce faire, nous ne voulions pas affecter le code métier existant mais nous voulions être capables

de tester plusieurs stratégies afin de trouver la plus efficace pour disséminer les données entre les pairs. Nous avons montré dans ce chapitre que le comportement des composants peut facilement varier en implémentant une fonction différemment dans l'API. Sur l'EventCloud, nous nous sommes principalement intéressés à deux des aspects d'une stratégie d'équilibrage de charge dont nous avons fait varier l'implémentation : comment est sélectionnée la charge à déplacer et est-ce que les pairs échangent de l'information avec d'autres pairs pour les aider à évaluer leur propre état de charge. Des différences significatives ont été observées en termes de distribution, surtout en ce qui concerne la façon de sélectionner la charge à déplacer.

Dans la seconde section de ce chapitre, nous avons présenté comment implémenter avec l'API générique la stratégie des fonctions de hachage variables présentée dans le Chapitre 4 dans un réseau CAN. Nous avons décrit diverses implémentations possibles pour calculer une nouvelle fonction de hachage et estimer l'état de charge d'un pair. Cela peut requérir ou non des connaissances externes sur la charge de certains pairs dans le système. Nous avons réalisé des expériences afin de valider notre approche et de la comparer avec une stratégie d'équilibrage de charge existante et connue qui consiste à ajouter un nouveau pair dans la zone d'un pair surchargé. Les résultats ont montré que les fonctions de hachage variables peuvent distribuer les données sur deux fois plus de pairs qu'avec cette stratégie célèbre, et 400 fois plus que si aucune stratégie n'est employée. En ce qui concerne le coût de notre solution, elle ne requiert pas plus de communication entre les pairs ni plus de transfert de données que des stratégies existantes et, contrairement à certaines d'entre elles, a l'avantage de ne pas changer la topologie du réseau (les mêmes liens de voisinage sont maintenus).
Par la suite, nous avons également réalisé des expériences pour montrer comment facilement améliorer ces performances en faisant varier le comportement de certains composants. En changeant quelques lignes de code, nous avons pu dans certains cas doubler le nombre de pairs stockant des données. Il est intéressant de voir à quel point certains paramètres, pourtant de faible importance à première vue, peuvent se révéler essentiels.

Enfin, nous avons implémenté la même stratégie des fonctions de hachage variables dans un réseau de type Chord. L'intérêt était de montrer que la même solution, et donc le même code, peut être déployé dans un système différent grâce à la généricité de l'API. Même si chaque système a ses propres spécificités, nous pensons que la séparation d'avec le reste du système qu'offre l'API peut faciliter le déploiement d'une stratégie existante vers un autre système. Les résultats avec Chord ont montré que les données peuvent aussi être distribuées convenablement mais à un coût plus élevé en termes de transfert de données et de temps écoulé avant d'arriver à l'équilibre, car il y a davantage de changements de fonctions de hachage initiés qu'avec CAN. Bien que nous ayons essayé de distribuer les données autant que possible alors que, dans un vrai système, il est parfois seulement nécessaire de décharger certaines machines, ces résultats laissent suggérer que la même stratégie d'équilibrage de charge peut ne pas avoir la même efficacité sur des systèmes différents. Par conséquent, il semble utile de pouvoir tester diverses stratégies dans un système donné afin de pouvoir choisir la plus efficace, ce qui est la finalité de l'API.

A.3 Conclusion

A.3.1 Résumé

Cette thèse a présenté des contributions autour du thème de l'équilibrage de charge dans les réseaux structurés. Ce travail a été influencé par le développement dans notre équipe de recherche de l'EventCloud, une plateforme distribuée pour le stockage et la récupération de données du Web Sémantique. Nous avons présenté une nouvelle solution basée sur l'utilisation de plusieurs fonctions de hachage préservant l'ordre naturel des données pour améliorer la dissémination de valeurs biaisées. Afin de pouvoir facilement implémenter et modifier cette stratégie, ainsi que n'importe quelle autre, une API générique a également été présentée.

Fonctions de hachage variables

Stocker des valeurs très biaisées comme celles des données du Web Sémantique de manière à préserver leur ordre naturel rend par conséquence la distribution de

ces valeurs parmi les pairs très biaisée elle aussi. Puisque les fonctions de hachage sont au cœur de la distribution des données dans les systèmes basés sur une DHT, nous proposons de nous attaquer à ce problème en autorisant un pair surchargé à modifier la fonction de hachage qu'il applique sur les données, afin de réduire l'intervalle de valeurs dont il est responsable. Il en résulte la cœxistence de plusieurs fonctions de hachage différentes utilisées par différents pairs. Afin de maintenir le routage optimal et la consistance du système, nous avons montré qu'il n'est pas nécessaire que tous les pairs utilisent la même fonction de hachage du moment que certaines règles sont respectées. A notre connaissance, ceci est la première solution d'équilibrage de charge pour réseaux structurés basée sur l'utilisation de plusieurs fonctions de hachage préservant l'ordre naturel et qui vise à améliorer la distribution des données. Pour évaluer l'efficacité de cette solution, nous avons simulé un système composé de 1000 pairs et injecté 1 000 000 triplets RDF aux valeurs fortement biaisées. Les résultats ont montré que le nombre de pairs stockant des données est de deux sur mille lorsqu'aucune stratégie n'est employée. Lorsque la solution des fonctions de hachage variables est utilisée, jusqu'à 945 pairs stockent des données à la fin des expériences. Nous avons également comparé notre stratégie avec une autre bien connue et qui consiste à ajouter un pair dans une zone gérée par un pair surchargé. Notre solution est apparue plus efficace pour répartir la charge et aussi moins coûteuse en ce qui concerne le volume de données à déplacer pour atteindre l'équilibre.

API générique

Implémenter une stratégie d'équilibrage de charge dans un système distribué n'est pas aussi simple que ce que l'on peut croire. Il existe un large éventail de solutions. Cependant, chaque système a sa propre architecture et aussi ses propres exigences de performance concernant différents critères. Ainsi, implémenter une stratégie existante peut ne pas donner les résultats escomptés puisque chaque système est différent et donc de meilleures solutions pourraient être implémentées et testées, par exemple en affinant une stratégie jusqu'à atteindre les performances voulues. Pour ce faire, nous avons présenté dans cette thèse une API générique pour implémenter l'équilibrage de charge dans les systèmes distribués. L'API a été conçue

suite à l'analyse de solutions existantes qui nous a permis d'identifier des critères de différenciation tels que *comment est évalué l'état de charge des nœuds ?* ou encore *comment est sélectionnée la charge à déplacer ?*. Ces critères représentent le bon niveau d'abstraction pour modéliser n'importe quelle stratégie et nous ont permis de définir des composants proposant un ensemble de fonctions bien définies. Modifier une stratégie consiste habituellement à modifier le comportement d'un ou quelques critères de différenciation. En utilisant notre API, cela consiste simplement à réimplémenter les fonctions correspondant à ces critères. Donc, cela n'affecte pas le code métier existant et, d'un point de vue développement, facilite les possibles changements par rapport aux modifications sur des stratégies codées en dur. Les expériences que nous avons effectuées pour implémenter, entre autres, la stratégie des fonctions de hachage variables ont montré que les résultats peuvent être nettement améliorés en fonction du comportement de certains composants. De plus, passer d'un comportement à un autre est trivial puisque cela consiste généralement à ré-implémenter une fonction de l'API, ce qui dans notre cas a nécessité de modifier seulement quelques lignes de code.

A.3.2 Perspectives

Nous avons réalisé nos expériences sur CAN et Chord mais nous pensons que l'utilisation de fonctions de hachage variables pourrait s'appliquer à d'autres systèmes basés sur une DHT. De même, nous avons uniquement utilisé des données RDF mais les mêmes principes pourraient être suivis avec n'importe quel type de données encodées en Unicode (par exemple, des paires clé-valeur ou des tuples). Nous avons présenté cette stratégie dans le contexte d'un système plaçant les données selon l'ordre lexicographique. D'autres types de tri pourraient être envisagés comme l'ordre croissant ou décroissant pour des valeurs numériques. Egalement, la ou les borne(s) associée(s) à chaque pair pourrait ne pas être basée sur la valeur des données mais sur leur format ou leur taille. Ainsi, il pourrait être possible d'utiliser cette technique pour des données non encodées en Unicode, tant que la fonction de hachage permet de retourner une coordonnée ou un identifiant pour localiser une donnée dans le réseau.
Nous avons montré que les fonctions de hachage variables peuvent aider à amélio-

rer la répartition des données. Concernant de possibles futures expérimentations, il serait peut-être intéressant d'évaluer à l'aide de cas d'utilisation les avantages de cette distribution améliorée sur la charge de traitement des pairs. Aussi, nous avons considéré comme notre principal critère de charge le nombre d'objets stockés par un pair pour estimer son état de charge mais un autre critère pourrait être utilisé : la popularité des données stockées par chaque pair. En effet, certains ensembles particuliers de données peuvent être fréquemment accédés ce qui génère de la charge de traitement pour le pair qui les stocke. Une alternative à la réplication serait d'autoriser un pair stockant peu d'objets mais surchargé à cause de la popularité de ces derniers à modifier sa fonction de hachage pour qu'une partie soit envoyée vers un ou plusieurs de ses voisins stockant potentiellement beaucoup d'objets mais avec une fréquence d'accès plus faible.
Enfin, dans cette thèse, nous avons principalement pris en compte l'équilibrage de charge pour les pairs *surchargés*. Cependant, la stratégie des fonctions de hachage variables pourrait aussi être étendue aux pairs *sous-chargés*. Le principe consisterait à autoriser un pair sous-chargé à modifier sa fonction de hachage afin d'agrandir l'intervalle de valeurs dont il est responsable.

Actuellement, nous envisageons d'explorer l'utilisation de fonctions de hachage variables dans le contexte du traitement de flux. Notre équipe de recherche travaille sur la construction d'une architecture distribuée pour l'analyse de flux RDF. Le résultat pourrait ressembler à ce qui est présenté sur la Figure A.1. Ce système considère des flux de données RDF produites par diverses sources comme des flux Twitter ou des réseaux de capteurs. Le but est de pouvoir évaluer en temps réel si ces données entrantes permettent de répondre à des requêtes (par exemple des souscriptions). Pour accélérer ce processus, une requête est divisée en sous-requêtes prenant la forme de triplets de telle sorte qu'elles puissent être exécutées en parallèle et que leurs résultats respectifs puissent être ensuite regroupés pour répondre à la requête initiale. Chaque sous-requête est associée avec un nœud *travailleur* responsable de tester la correspondance entre les triplets entrants et celle-ci. Pour traiter efficacement les flux et les requêtes, chacun de ces nœuds est exclusivement responsable de triplets et de sous-requêtes contenant une valeur de prédicat bien définie. Les flux de données sont reçus aléatoirement par des machines de la

Couche 1, puis envoyés aux nœuds de répartition de la Couche 2 qui pourraient être organisés en un réseau structuré de type Chord ou skip list. Chaque nœud de répartition serait responsable d'un certain intervalle de valeurs de prédicat et recevrait donc tous les triplets entrants associés à cet intervalle (sur la Figure A.1, la lettre écrite à l'intérieur de ces nœuds représente la borne maximale du nœud). Enfin, les nœuds de répartition enverraient aux travailleurs (Couche 3) les triplets contenant la valeur de prédicat dont ils sont responsable afin de commencer à évaluer la correspondance triplets/sous-requête et de finalement combiner les résultats.

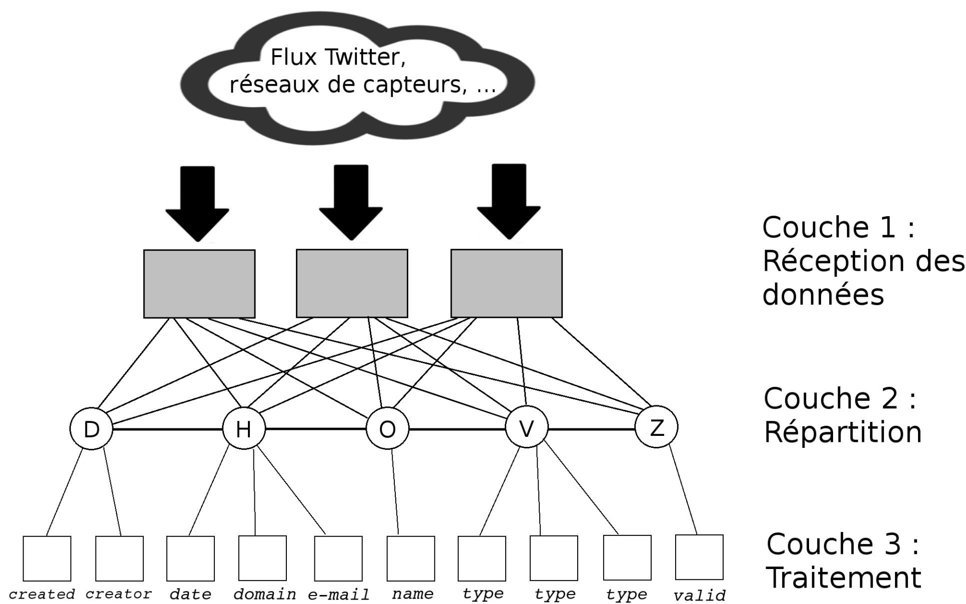


Figure A.1 – Architecture envisagée pour le traitement de flux RDF.

Nous envisageons la possibilité d'implémenter une stratégie d'équilibrage de charge sur ce futur système afin d'empêcher que des nœuds travailleurs ou de répartition soient surchargés par des flux ou même par trop de sous-requêtes ayant le même prédicat populaire. Puisque l'un des principaux avantages des fonctions de hachage variables est d'aider à équilibrer la charge entre les nœuds pour s'adapter en temps réel aux données entrantes, cette stratégie pourrait être envisagée comme une possible solution dans ce contexte. Par exemple, si un nœud de la Couche 2 reçoit trop de triplets car il gère l'intervalle des prédicats les plus populaires, celui-

ci pourrait modifier sa fonction de hachage afin de recevoir moins de triplets et donc de réduire sa charge de traitement (puisque nous supposons que les données ne sont pas stockées). Sinon, d'autres solutions d'équilibrage de charge pourraient être testées avec notre API générique, comme celle proposée par Nasir *et al.* [47] (choix du nœud le moins chargé pour recevoir le flux), Meghdoot [54] (ajout de nœuds aux endroits surchargés par les souscriptions) ou même une toute nouvelle solution conçue selon les principes proposés avec l'API générique.

Bibliography

[1] Maeva Antoine and Fabrice Huet. Dealing with skewed data in structured overlays using variable hash functions. In *Parallel and distributed computing, applications and technologies (pdcat), 2014 15th international conference on.* (Hong Kong). IEEE Computer Society Washington, DC, USA, 2014, pages 42–48. DOI: 10.1109/PDCAT.2014.15 (cited on pp. 4, 145).

[2] Maeva Antoine, Laurent Pellegrino, Fabrice Huet, and Françoise Baude. A generic api for load balancing in structured p2p systems. In *Computer architecture and high performance computing workshop (sbac-padw), 2014 international symposium on.* (Paris). IEEE Computer Society Washington, DC, USA, 2014, pages 138–143. DOI: 10.1109/SBAC-PADW.2014.17 (cited on pp. 4, 145).

[3] Maeva Antoine, Laurent Pellegrino, Fabrice Huet, and Françoise Baude. A generic api for load balancing in distributed systems for big data management. *Concurrency and computation: practice and experience*, Wiley Online Library, 2015. DOI: 10.1002/cpe.3646 (cited on pp. 4, 146).

[4] Iyad Alshabani, Maeva Antoine, Françoise Baude, Fabrice Huet, and Laurent Pellegrino. Eventcloud version 1. *Dépôt APP IDDN FR.001.470019.000.S.P.2014.000.31235*, 2014 (cited on p. 8).

[5] Laurent Pellegrino. Pushing dynamic and ubiquitous event-based interaction in the Internet of services: a middleware for event clouds. French. PhD Thesis. University of Nice Sophia Antipolis, April 2014. URL: http://tel.archives-ouvertes.fr/tel-00984262 (cited on p. 8).

[6] Ben Adida, Ivan Herman, Manu Sporny, and Mark Birbeck. RDFa 1.1 primer: rich structured data markup for Web documents, 2012. URL: http://www.w3.org/TR/xhtml-rdfa-primer/ (visited on 06/24/2015) (cited on pp. 8, 9).

[7] Ora Lassila and Ralph R Swick. Resource description framework (RDF) model and syntax specification, 1999 (cited on pp. 9, 19, 32).

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: a nucleus for a web of open data. In, *The semantic web*, pages 722–735. Springer, 2007 (cited on pp. 9, 116).

[9] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media meets semantic web–how the bbc uses dbpedia and linked data to make connections. In, *The semantic web: research and applications*, pages 723–737. Springer, 2009 (cited on p. 10).

[10] Spyros Kotoulas, Eyal Oren, and Frank Van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. In *Proceedings of the 19th international conference on world wide web*. ACM, 2010, pages 531–540 (cited on pp. 11, 32, 55, 79, 146).

[11] Jeremy J Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the international conference on World Wide Web*. ACM, 2005, pages 613–622 (cited on p. 11).

[12] Eric Prud'Hommeaux and Andy Seaborne. SPARQL query language for RDF. *W3C recommendation*, 15, 2008. URL: http://www.w3.org/TR/rdf-sparql-query (visited on 06/24/2015) (cited on p. 11).

[13] Stuart Weibel, John Kunze, Carl Lagoze, and Misha Wolf. Dublin core metadata for resource discovery. *Internet Engineering Task Force RFC*, 2413:222, 1998 (cited on p. 12).

[14] Bram Cohen. The BitTorrent protocol specification. 2008 (cited on p. 13).

[15] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*. Volume 7, 2007, pages 205–220 (cited on p. 13).

[16] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@Home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002 (cited on p. 13).

[17] Gnutella. URL: `http://rfc-gnutella.sourceforge.net/` (visited on 06/24/2015) (cited on p. 13).

[18] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*. Volume 31. (4). ACM, 2001, pages 149–160 (cited on pp. 14, 49, 53, 125).

[19] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*. Volume 31(4). ACM, 2001 (cited on pp. 14, 19, 49, 51).

[20] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In, *Peer-to-Peer Systems*, pages 53–65. Springer, 2002 (cited on pp. 14, 49, 52).

[21] Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*. Springer, 2001, pages 329–350 (cited on pp. 14, 49).

[22] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the ACM symposium on Theory of Computing*. ACM, 1997, pages 654–663 (cited on p. 14).

[23] Prasanna Ganesan, Krishna Gummadi, and Hector Garcia-Molina. Canon in g major: designing dhts with hierarchical structure. In *Distributed computing systems, 2004. proceedings. 24th international conference on*. IEEE, 2004, pages 263–272 (cited on p. 14).

[24] Dmitry Korzun and Andrei Gurtov. Survey on hierarchical routing schemes in "flat"distributed hash tables. *Peer-to-Peer Networking and Applications*, 4(4):346–375, 2011 (cited on p. 14).

[25] Bin Shao, Haixun Wang, and Yatao Li. The Trinity graph engine. *Microsoft Research*, 2012. URL: `http://research.microsoft.com/pubs/161291/trinity.pdf` (visited on 06/24/2015) (cited on p. 14).

[26] Imen Filali, Laurent Pellegrino, Francesco Bongiovanni, Fabrice Huet, and Françoise Baude. Modular P2P-based approach for RDF data storage and retrieval. In *Proceedings of the international conference on Advances in P2P Systems*, 2011 (cited on p. 23).

[27] Andy Seaborne. Jena TDB. 2009. URL: `http://jena.apache.org/documentation/tdb` (visited on 03/18/2015) (cited on pp. 23, 26).

[28] Ludovic Henrio, Fabrice Huet, and Justine Rochas. An optimal broadcast algorithm for content-addressable networks. In, *International conference On Principles of Distributed Systems (OPODIS)*, 2013 (cited on pp. 24, 68).

[29] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of the international conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2005, pages 1216–1227 (cited on p. 26).

[30] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: a generic architecture for storing and querying rdf and rdf schema. In, *The Semantic Web — ISWC*, pages 54–68. Springer, 2002 (cited on p. 26).

[31] Stephen Harris and Nicholas Gibbins. 3store: efficient bulk rdf storage, 2003 (cited on p. 26).

[32] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Very Large Data Base (VLDB)*. VLDB Endowment, 2007, pages 411–422 (cited on p. 26).

[33] Avinash Lakshman and Prashant Malik. Cassandra: a structured storage system on a P2P network. In *Proceedings of the symposium on Parallelism in Algorithms and Architectures*. ACM, 2009, pages 47–47 (cited on pp. 26, 50).

[34] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011 (cited on pp. 26, 50).

[35] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly, 2013 (cited on p. 27).

[36] Jans Aasman. Allegro graph: RDF triple database. Technical report. Franz Incorporated, 2006. URL: `http://www.franz.com/agraph/allegrograph/` (visited on 06/25/2015) (cited on p. 27).

[37] Günter Ladwig and Andreas Harth. CumulusRDF: linked data management on nested key-value stores. In *International workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2011, page 30 (cited on p. 27).

[38] Imen Filali, Francesco Bongiovanni, Fabrice Huet, and Françoise Baude. A survey of structured p2p systems for rdf data storage and retrieval. In, *Transactions on large-scale data-and knowledge-centered systems iii*, pages 20–55. Springer, 2011 (cited on p. 27).

[39] Peter Haase, Jeen Broekstra, Marc Ehrig, Maarten Menken, Peter Mika, Mariusz Olko, Michal Plechawski, Pawel Pyszlak, Björn Schnizler, Ronny Siebes, Steffen Staab, and Christoph Tempich. Bibster: a semantics-based bibliographic peer-to-peer system. In, *The Semantic Web - ISWC*, pages 122–136. Springer, 2004 (cited on p. 27).

[40] Jing Zhou, Wendy Hall, and David De Roure. Building a distributed infrastructure for scalable triple stores. *Journal of Computer Science and Technology*, 24(3):447–462, 2009 (cited on p. 27).

[41] Min Cai and Martin Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the international conference on World Wide Web*. ACM, 2004, pages 650–657 (cited on pp. 27, 37, 55, 126).

[42] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. MAAN: a multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004 (cited on pp. 27, 55).

[43] Dominic Battré, Felix Heine, André Höing, and Odej Kao. On triple dissemination, forward-chaining, and load balancing in DHT based RDF stores. In *Proceedings of the international conference on Databases, Information Systems, and Peer-to-Peer Computing*. Springer-Verlag, 2005, pages 343–354 (cited on pp. 27, 38).

[44] Akiyoshi Matono, Said Mirza Pahlevi, and Isao Kojima. RDFCube: a P2P-based three-dimensional index for structural joins on distributed triple stores. In, *Databases, Information Systems, and Peer-to-Peer Computing*, pages 323–330. Springer, 2007 (cited on p. 28).

[45] MongoDB. URL: `http://www.mongodb.org/` (visited on 01/15/2015) (cited on p. 33).

[46] Apache Storm. URL: `https://storm.apache.org/` (visited on 01/15/2015) (cited on p. 34).

[47] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: practical load balancing for distributed stream processing engines. In *31st international conference on data engineering (icde)*. IEEE, 2015, pages 137–148 (cited on pp. 34, 139, 156).

[48] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001 (cited on pp. 34, 53).

[49] Amazon Web Services. URL: `https://aws.amazon.com/` (visited on 01/15/2015) (cited on p. 35).

[50] Pascal Felber, Peter Kropf, Eryk Schiller, and Sabina Serbu. Survey on load balancing in peer-to-peer distributed hash tables. *Communications surveys & tutorials, ieee*, 16(1):473–492, 2014 (cited on pp. 36, 37).

[51] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured P2P systems. In, *Peer-to-Peer Systems II*, pages 68–79. Springer, 2003 (cited on p. 39).

[52] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in dynamic structured P2P systems. In *Conference of the IEEE Computer and Communications Societies*. Volume 4. IEEE, 2004, pages 2253–2262 (cited on p. 41).

[53] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, load balancing and efficient range query processing in dhts. In, *Advances in database technology-edbt 2006*, pages 131–148. Springer, 2006 (cited on pp. 41–43, 126).

[54] Abhishek Gupta, Ozgur D Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the international conference on Middleware*. Springer-Verlag New York, Inc., 2004, pages 254–273 (cited on pp. 43, 105, 117, 139, 156).

[55] Ashwin Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *Computer Communication Review*, 34(4):353–366, 2004 (cited on pp. 45, 126).

[56] Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. Fast and cost-effective online load-balancing in distributed range-queriable systems. *Parallel and distributed systems, ieee transactions on*, 22(8):1350–1364, 2011 (cited on pp. 46, 47, 113).

[57] James Aspnes and Gauri Shah. Skip graphs. *Acm transactions on algorithms (talg)*, 3(4):37, 2007 (cited on p. 46).

[58] M Seltzer. Oracle nosql database. *Oracle white paper*, 2011 (cited on p. 50).

[59] Ye Xia, Shigang Chen, Chunglae Cho, and Vivekanand Korgaonkar. Algorithms and performance of load-balancing with multiple hash functions in massive content distribution. *Computer networks*, 53(1):110–125, 2009 (cited on p. 51).

[60] Yuqi Mu, Cuibo Yu, Tao Ma, Chunhong Zhang, Wei Zheng, and Xiaohua Zhang. Dynamic load balancing with multiple hash functions in structured p2p systems. In *Wireless communications, networking and mobile computing, 2009. wicom'09. 5th international conference on*. IEEE, 2009, pages 1–4 (cited on p. 52).

[61] Tai-Ting Wu and Kuochen Wang. An efficient load balancing scheme for resilient search in kad peer to peer networks. In *Communications (micc), 2009 ieee 9th malaysia international conference on*. IEEE, 2009, pages 759–764 (cited on p. 52).

[62] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In, *Peer-to-Peer Systems II*, pages 80–87. Springer, 2003 (cited on p. 53).

[63] Karl Aberer, Philippe Cudré-Mauroux, Manfred Hauswirth, and Tim Van Pelt. Gridvine: building internet-scale semantic overlay networks. In, *The semantic web–iswc 2004*, pages 107–121. Springer, 2004 (cited on p. 55).

[64] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: a distributed, searchable key-value store. *Acm sigcomm computer communication review*, 42(4):25–36, 2012 (cited on p. 55).

[65] Yingwu Zhu and Yiming Hu. Efficient, proximity-aware load balancing for dht-based p2p systems. *Parallel and distributed systems, ieee transactions on*, 16(4):349–361, 2005 (cited on p. 55).

[66] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the acm*, 24(1):9–17, 1981 (cited on p. 72).

[67] Jing Gao, Wei Fan, Jiawei Han, and S Yu Philip. A general framework for mining concept-drifting data streams with skewed distributions. In *Sdm*. SIAM, 2007, pages 3–14 (cited on pp. 79, 146).

[68] Clifton Phua, Damminda Alahakoon, and Vincent Lee. Minority report in fraud detection: classification of skewed data. *Acm sigkdd explorations newsletter*, 6(1):50–59, 2004 (cited on pp. 79, 146).

[69] George Heineman and William T Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, 2001 (cited on p. 88).

[70] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the ieee/acm international workshop on grid computing*. IEEE Computer Society, 2005, pages 99–106 (cited on p. 104).

[71] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. *Acm sigplan notices*, 33(10):36–44, 1998 (cited on p. 107).

[72] Noel DePalma, Konstantin Popov, Nikos Parlavantzas, Per Brand, and Vladimir Vlassov. Tools for architecture based autonomic systems. In *Icas*. IEEE, 2009, pages 313–320 (cited on p. 108).

[73] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the international symposium on High-performance Parallel and Distributed Computing*. ACM, 2013, pages 115–116 (cited on p. 108).

[74] Alberto Montresor and Márk Jelasity. Peersim: a scalable p2p simulator. In *Peer-to-peer computing, 2009. p2p'09. ieee ninth international conference on*. IEEE, 2009, pages 99–100 (cited on p. 114).

[75] Christian Bizer and Andreas Schultz. The berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009 (cited on p. 116).

[76] Luigi Liquori, Cédric Tedeschi, and Francesco Bongiovanni. Babelchord: a social tower of dht-based overlay networks. In *Computers and communications, 2009. iscc 2009. ieee symposium on*. IEEE, 2009, pages 307–312 (cited on p. 126).

List of Acronyms

AWS Amazon Web Services . . . 35
CAN Content Addressable Network . . . 14
DHT Distributed Hash Table . . . 4
EC EventCloud . . . 8
IRI Internationalized Resource Identifier . . . 9
MAAN Multiple Attribute Addressable Network . . . 27
NoSQL Not only SQL . . . 2
OWL Web Ontology Language . . . 10
P2P Peer-to-Peer . . . 2
RDBMS Relational Database Management Systems . . . 24
RDF Resource Description Framework . . . 9
RDFS RDF Schema . . . 10
SON Structured Overlay Network . . . 14
SONs Structured Overlay Networks . . . 2
SPARQL SPARQL Protocol and RDF Query Language . . . 11
URI Uniform Resource Identifier . . . 9