UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS

École Doctorale STIC

Sciences et Technologies de l'Information et de la Communication

THÈSE

pour l'obtention du titre de

Docteur en Sciences

Mention Informatique

présentée et soutenue par

Laurent Pellegrino

Pushing dynamic and ubiquitous event-based interactions in the Internet of services: a middleware for event clouds

Thèse dirigée par Françoise Baude
et co-encadrée par Fabrice Huet

Soutenue le 3 Avril 2014

Jury

Rapporteurs	Ioana Manolescu	Inria Saclay - Île-de-France
	Etienne Rivière	Université de Neuchâtel
Examinateurs	Johan Montagnat	CNRS
	Ester Pacitti	Université de Montpellier 2
Directeur de thèse	Françoise Baude	Université de Nice-Sophia Antipolis
Co-encadrant de thèse	Fabrice Huet	Université de Nice-Sophia Antipolis

À mes parents et ma grande soeur,

Résumé

Resource Description Framework (RDF) est devenu un modèle de données pertinent afin de décrire et de modéliser les informations qui sont partagées sur le Web. Cependant, fournir une solution permettant de stocker et de récupérer ces données de manière efficace tout en passant à l'échelle reste un défi majeur. Dans le contexte de cette thèse nous proposons un intergiciel dévoué au stockage, à la récupération synchrone mais aussi à la dissémination sélective et asynchrone en quasi temps réel des informations de type RDF dans un environnement complètement distribué. L'objectif est de pouvoir tirer parti des informations du passé comme de celles filtrées en quasi temps réel. Dans ce but, nous avons construit notre système sur une version légèrement modifiée du réseau Pair-à-Pair CAN à 4-dimensions afin de refléter la structure d'un n-uplet RDF. Contrairement à une grande majorité de solutions existantes, nous avons avons fait le choix d'éviter le hachage pour indexer les données ce qui nous permet de traiter les requêtes à intervalles de manière efficace mais aussi de soulever des défis techniques intéressants. Le filtrage des informations en quasi temps réel est permis par l'expression des intérêts à l'aide de souscriptions basées sur le contenu des évènements futurs. Les souscriptions sont traitées par une couche publish/subscribe conçue sur l'architecture CAN. Nous avons proposé deux algorithmes qui permettent de vérifier la concordance des évènements RDF avec les souscriptions enregistrées et de les transférer vers les entités intéressées lorsque la correspondance se vérifie. Les deux algorithmes ont été testés expérimentalement en termes de débit d'évènements par seconde et de passage à l'échelle. Bien que l'un permet de meilleures performances que l'autre, ils restent complémentaires pour s'assurer que tout évènement soit notifié s'il doit l'être. En sus de la récupération synchrone et de la diffusion asynchrone d'évènements, nous nous sommes intéressés à améliorer, avec notre système, la répartition des données RDF qui souffrent de dissymétrie. Finalement, nous avons consacré un effort non négligeable à rendre notre intergiciel modulaire. Cela a permis d'améliorer sa maintenance et sa réutilisabilité puisque l'architecture modulaire réduit le couplage entre les différents composants qui le constitue.

Abstract

RDF has become a relevant data model for describing and modeling information on the Web but providing scalable solutions to store and retrieve RDF data in a responsive manner is still challenging. Within the context of this thesis we propose a middleware devoted to storing, retrieving synchronously but also disseminating selectively and asynchronously RDF data in a fully distributed environment. Its purposes is to allow to leverage historical information and filter data near real-time. To this aims we have built our system atop a slightly modified version of a 4-dimensional Content Addressable Network (CAN) overlay network reflecting the structure of an RDF tuple. Unlike many existing solutions we made the choice to avoid hashing for indexing data, thus allowing efficient range queries resolution and raising interesting technical challenges. Near realtime filtering is enabled by expressing information preferences in advance through content-based subscriptions handled by a publish/subscribe layer designed atop the CAN architecture. We have proposed two algorithms to check RDF data or events satisfaction with subscriptions but also to forward solutions to interested parties. Both algorithms have been experimentally tested for throughput and scalability. Although one performs better than the other, they remain complementary to ensure correctness. Along with information retrieval and dissemination, we have proposed a solution to enhance RDF data distribution on our revised CAN network since RDF information suffers from skewness. Finally, to improve maintainability and reusability some efforts were also dedicated to provide a modular middleware reducing the coupling between its underlying software artifacts.

Acknowledgments

I would like to thank Françoise and Fabrice for the opportunity they gave me to make this thesis a reality but also for their help throughout these three years and more. Also, I would like to express my appreciation to Ioana Manolescu and Etienne Rivière for agreeing to review this thesis but also Johan Montagnat for doing me the honor to preside my jury.

Undeniably, I have to thank several coworkers. I think especially to Imen and Francesco that have trained me in the research world. My thanks also go to Bastien, Iyad, Maeva, Justine and all great persons from the SCALE and former OASIS team but also all the people I worked with or met during this thesis.

I want also thank my family. First, all my gratitude goes to my parents who have encouraged me during this period. I also thank my brother in law and my sister, who despite the difficult facts of life are always in a good mood and share their joy with others and especially me. Finally, my thanks go to Pauline aka Popo, my great-niece, without whom this thesis would probably never come to an end.

Table of Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

In recent years the Internet traffic exchanged has grown exponentially. It is explained by the amount of information generated by users and new services that thrive. As stated by Eric Schmidt in 2010, every two days now we create as much information as we did from the dawn of civilization up until 2003. At this time the amount of information was already something like five exabytes of data, he said. Besides, with the advent of the Internet of things, a concept that refers to uniquely identifiable objects that communicate over the Internet, we are probably just at the beginning of this exponential growth of information. For example, Cisco expects the global IP traffic will reach 1.4 zettabytes per year in 2017[1].

This explosion regarding information exchanged gave birth to a new field of computer science called Data Mining. The overall goal of data mining process is to discover interesting patterns in large datasets. A concrete example is the

[1] `http://goo.gl/dj85Ul`

Google Knowledge Graph [1] that provides additional valuable information when you perform a search. A more recent illustration is the scandal caused by the Prism surveillance program operated by the United States National Security Agency (NSA) whose the purpose is to collect and correlate meta-data from users, unknown to them, to prevent terrorist acts.

A prerequisite to Data Mining is to aggregate streams of interest and store incoming data for analysis. These bases of knowledge are realized by data warehouses that are huge repositories of data which are created by integrating data from one or more disparate sources. When sources are outputs provided by any actor publishing heterogeneous information over the Internet, building data warehouses raises one main question: how to filter information of interest and correlate them with others? A key element to answer this question consists of using structured data to make information machine processable and machine understandable.

The Semantic Web movement has generated tremendous interest these last years. It aims to turn the Web, i.e. Web documents, into a gargantuan database where computers could fetch data in an homogeneous manner. The interesting point here is that the Semantic Web community already provides a full technology stack (RDF, SPARQL, RDFS, OWL, etc.) to address most of the issues related to the previous question, but mainly in a synchronous and centralized[2] environment. Within the context of the European PLAY project, one of the projects in which this thesis has been developed, we are investigating how we can leverage the Semantic Web representation model and thus the existing stack to filter, detect and react on interesting patterns or situations. In this context, the purpose of the PLAY project is to bring an elastic and reliable architecture for dynamic and complex, event-driven interaction in highly distributed and heterogeneous service systems. Such an architecture will enable ubiquitous exchange of information between heterogeneous services, providing the possibilities to adapt and personalize their execution, resulting in the so-called situational-driven process adaptivity.

A typical example of the scenario we are trying to achieve with the PLAY project is illustrated by the following motivating story. Let's say that Paul is a

[2]In the rest of this thesis, we refer to centralized for solutions that concentrate under a single location/front end, even if behind, several machines or physical resources are involved like it is the case with the Master-Slave approach where a master receives and dispatches requests to multiple slaves.

businessman who has been flying from Paris to New York. He used the entertainment service on board, but hasn't finished watching the movie before the landing. Two hours later he is entering his room in the downtown hotel he booked earlier and wow: the room entertainment service is ready to PLAY the movie Paul was watching in the plane – of course starting where he left it.

Realizing such a scenario raises several technical questions which some are introduced in the next section and addressed throughout this thesis.

1.2 Problem Definition

In this thesis we are focusing on two key problematics that can be summarized by the following two questions: *How can we efficiently store and retrieve Semantic Web data in a distributed environment? How can we pragmatically filter and disseminate Semantic Web events to users with individual preferences?*

The inherent scalability issue that arises with distributed systems has mainly be addressed the last years by resorting to Peer-to-Peer (P2P) networks that avoid a single point of access. However, relying on the Semantic Web model, depicted by RDF, raises several challenges which have a direct impact on the underlying network topology that is considered.

The first challenge stems from the expressivity level of the common SPARQL language that is usually used to retrieve RDF data. Some may have noticed the lexicographic similarity with SQL. The analogy is not a coincidence. SPARQL is a query language for RDF data modeled long after SQL. Although they are quite different since they do not achieve the exact same purpose, SPARQL supports very complex operators that makes it as expressive or even more expressive than SQL. In this manuscript, we will see how the data model and query language affect many of the choices we made such as the design of the P2P architecture, the routing algorithms and the storage of RDF data.

The second challenge is related to the filtering of RDF data from publishers to interested parties. As explained in the motivation section, the scenario envisaged within the PLAY project is mainly based on data-driven querying [2] that focus on near real-time conditions that should be satisfied. A prerequisite is to filter information of interest but not only. Events have to be stored to act as additional

context and to perform a kind of analytic on the past data. Here, Semantic Web data have, again, a significant impact on how the Publish/Subscribe layer must be designed. With an additional complexity since in event based systems the entities are loosely coupled. Communications are done in an asynchronous manner, thus providing less guarantees than the traditional request/reply message exchange pattern (e.g. lack of delivery guarantee). In this context, we will see how RDF data may be represented as events and how they differ from traditional multi-attribute events. In addition the combination between event filtering and the storage requirement triggers several questions about efficiency and consistency: e.g., how to ensure operations ordering from a same client? which kind of throughput can we expect? how to ensure that events are stored once they are delivered? these questions will be discussed and addressed.

Finally, a third challenge we take care is about load balancing and the elasticity property of modern distributed systems we intend to leverage in order to ensure a certain level of performance. We will see that the choice we made, that consists of removing the use of hash functions to exploit more complex queries or subscriptions than simple exact matching exposes us to load imbalances. In real scenarios, any dataset is skewed, but here the imbalance is accentuated by one of the characteristics of RDF data that implies some values to share common prefixes.

1.3 Outline and Contributions

The major contribution of this thesis is the definition and the implementation of a modular middleware for storing, retrieving and disseminating RDF data and events in cloud environments. It is structured around three major works organized in three dedicated chapters whose the content is summarized along with others hereafter:

- **Chapter 2** gives an overview of the main concepts and technologies we refer to throughout this thesis. First, we introduce the Peer-to-Peer paradigm. Then, we draw attention to the Semantic Web and discuss the main benefits of using semantic before focusing on the Publish/Subscribe communication

style. Finally, we detail the ProActive middleware, which is the main technology used for implementing the middleware developed in this thesis.

- **Chapter 3** presents our first contribution that relates to a distributed RDF storage infrastructure which was introduced in [3] and awarded Best Paper of the AP2PS 2011 conference. A brief related works section about decentralized systems for storing and retrieving RDF data introduces this chapter. Then, we introduce the popular CAN P2P protocol which is the underlying P2P overlay network we rely on for routing messages and, indirectly, for achieving scalability. Afterwards, we motivate and discuss the design choices and the adjustments we made regarding the CAN protocol before explaining in a second section how messages are routed with our modifications. In a penultimate section we describe in details how RDF data is indexed in the P2P network and how SPARQL queries are executed. Finally, we provide the results we got by experimenting our solution on the Grid'5000 testbed.

- **Chapter 4** enters into the details of our second contribution that relates to a publish/subscribe layer for storing and disseminating RDF events. It is built as an extension atop the infrastructure introduced in the previous chapter and relies on the routing algorithms that were described earlier. We start to compare existing solutions and we explain why building RDF-based event systems differs from traditional publish/subscribe systems. Then, we introduce our publish/subscribe infrastructure for RDF events. First, we detail the event and subscription model suitable for RDF data we propose. Afterwards we list the different properties our publish/subscribe system is assumed to respect, before entering into the details of two publish/subscribe algorithms. Their characteristics and differences are explained, discussed and analyzed. To conclude, the algorithms we propose are evaluated in a distributed environment with up to 29 machines. This second contribution has been accepted and presented at Globe 2013 [4].

- **Chapter 5** highlights our third contribution which is about load balancing with RDF data. The first section summarizes how load balancing solutions have evolved over the time and what are existing solutions to fix load im-

balances with RDF data but not only. Then, we describe our solution by explaining the different choices that are conceivable and the ones we have opted for. As it will be explained, our approach combines standard mechanisms such as online statistical information recording and gossip protocols for exchanging load information. In a last section we discuss the results obtained for the empirical evaluations we have made with real data.

- **Chapter 6** gives an overview of the EventCloud middleware, which is the middleware developed within the context of this thesis. The purpose of this chapter is to give an overview of the system from an architectural and implementation point of view. In a first step, we highlight the different components that make up the system. Then, we summarize the different features and show how flexible and modular the middleware is since it has been built with clear separations between the basic subcomponents of the whole API. In particular we see how modularity plays a significant role in our proposed architecture and what kind of advantages it brings regarding the components which form our infrastructure. Then, we focus on some implementation details we have faced up to and addressed to make the system efficient and responsive.

- **Chapter 7** concludes the thesis. It reviews the contributions and presents some research and development perspectives that may raise from this thesis.

Finally, regarding the contributions made within the context of this thesis we can also notice that the EventCloud middeware has been tested and validated with the different scenarios created in the PLAY project [5, 6, 7, 8, 9, 10, 11]. Besides, the EventCloud middleware has also been used and evaluated in other contexts. For example by providing building blocks to distribute Complex Event Processing (CEP) engines that aim to correlate multiple real-time events and past events [12]. Another application relates to lazy data transfers where events embed large attachments that do not need to transit through the event service. Only event descriptions are conveyed to the EventCloud before being disseminated to interested parties. The attachments are transfered in a lazy and transparent manner by enabling direct publisher to subscriber data exchange [13].

Chapter 2

Background

Contents

In this chapter we introduce and give an overview of the main concepts and technologies we refer to throughout this thesis. First we present the Peer-to-Peer paradigm. Then, we focus on the Publish/Subscribe communication style before

entering into the details of the Semantic Web movement and discuss the main benefits of using semantic. Finally, we detail the ProActive middleware, which is the main technology used for implementing the architecture and the algorithms introduced in the next chapters.

2.1 The Peer-to-Peer Paradigm

P2P systems have generated tremendous interest the last 15 years and are now recognized as a key communication model to build large scale distributed applications [14]. This model differs from the traditional *client/server* approach where client nodes request resources provided by a central server. With the P2P model, all the machines or nodes (also called peers) play the same role. Each peer acts both as a client and server. It can share its resources with other peers and make use of those provided by some others. The resources are sources or supplies from which benefits are produced. They can be of different types such as data, URLs, files, disk storage, bandwidth, CPU cycles, etc. Moreover, since all nodes are suppliers, the overall aggregated system capacity is increased compared to a client/server model.

2.1.1 P2P overlays

In a P2P network the nodes are self-organized into an overlay network that runs atop a physical network topology. The virtual topology of overlay networks allows to build and deploy distributed services without having to modify the IP protocols. In addition, unlike the client/server model, the peers communicate with each other without any centralized coordination. As a result, the full decentralization of P2P overlays makes them, generally, scale with respect to the number of nodes in the network. This scalability property is one of the most prominent features of P2P systems and explains the attractiveness of the model along with its built-in fault tolerance, replication and load balancing properties to adapt to the arrival, departure and failure of peers.

P2P overlays are usually classified into three main categories: *unstructured*, *structured* and *hierarchical*; based on the topology construction techniques. Fig-

ure 2.1 gives an overview of the three main type of overlays we introduce in the following subsections.

Unstructured overlays

Unstructured P2P systems belong to the first generation of P2P overlays that appeared with the P2P model. They are characterized by the absence of constraints regarding data placement and links establishment between peers. The resources are indexed on peers at random and peers connect to each other in an arbitrary manner. This lack of organization and structure implies to flood the whole network or to use heuristics to lookup a resource, thus leading to scalability issues or no solution for unpopular resources. On the other hand unstructured overlays are very resilient to peers arrival and departure (phenomenon known as *churn*) and that may explain their success in some domains like file sharing and streaming.

To summarize, unstructured P2P systems are often really simple to set up and implement but provide limited guarantees, not to say no guarantee, on search operations. Systems based on this type of overlay are numerous [15, 16, 17, 18]. Gnutella [15] is one of the first unstructured P2P network that has brought out the problem of plenty of communication between peers. For example, in its version 0.4, the volume of data that relates to the protocol was as important as the one generated by information exchanged as requested by end users [19].

Structured overlays

The drawbacks of unstructured P2P overlays have been intensively studied and these efforts gave birth in 2001, with CAN [20], Chord [21] and Pastry [22], to a new type of overlay called Structured Overlay Network (SON). SONs strive to solve the issues that occur with their unstructured opposite by providing an upper bound limit on the number of messages required to find a resource in the network. This is made possible by organizing the peers in well known geometrical topologies (hyper-cube, ring, tree, etc.) that provide interesting mathematical properties. In return, structured overlays incur a small overhead to maintain a consistent view of the geometrical structure among peers under churn. However, this cost is most of the time negligible with regards to the benefits they supply.

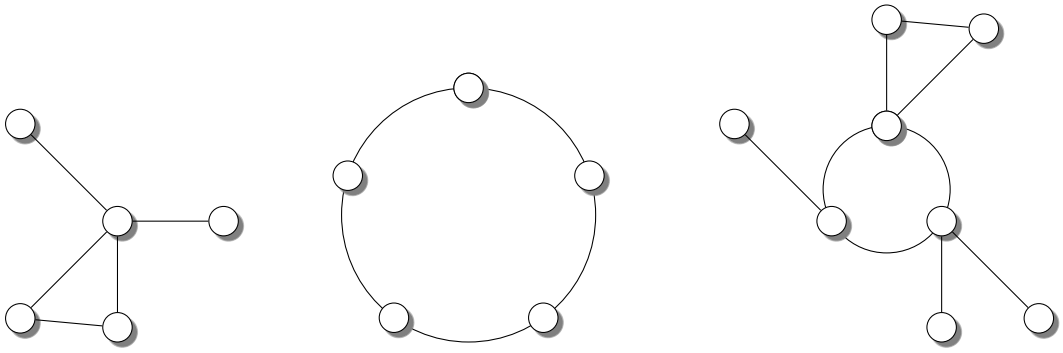
Usually, structured P2P protocols are provided with a standard abstraction called Distributed Hash Table (DHT) that offers a simple API similar to hash tables. It consists of two main primitives *put(key, value)* and *get(key)* that allow respectively to store and fetch data by key. The first generation of DHTs makes use of consistent hashing to map a key to a value and the resulting binding to a node. Briefly speaking, consistent hashing ensures that only a fraction of keys have to be remapped to nodes when some are joining and leaving the system. Moreover it eliminates the occurrence of hot spots and prevent nodes from becoming swamped by balancing the load between peers uniformly with high probability [23]. From an architectural point of view, each peer is assigned a part of a global space identifier like a circle and is responsible for all the keys that fall in its range. Then, consistent hash functions such as MD5 or SHA1 are used to associate a key to a value. Despite to the fact that the DHT abstraction is really well suited for manipulating key/value pairs, it supports only exact matching and not complex queries such as conjunctives and range queries. Some research efforts have been made in this way [24, 25] but it remains an ongoing area of research depending of the consistency, the availability, the data model but also the use case the considered system must deal with.

Hierarchical overlays

With their success, the design of distributed systems tends to grow in terms of complexity and requires more intelligence and processing in routing. Hierarchical or hybrid overlays try to fill this lack by exploiting the properties of multiple structured and/or unstructured P2P overlays. The topology consists of several nodes from two or more type of overlays that are organized into groups and groups are interconnected to form a connected graph. Each group depicts a layer that has its own purpose and applies its own routing mechanisms.

P2P systems that are mentioned as hybrid overlays or that refer to super peers may be seen as a particular case of hierarchical overlays with two groups. For example in the file sharing context, one upper level group acts as an index to locate available resources. Then, resources are exchanged by contacting peers from the second lower group that effectively contain the resources. Examples of

hierarchical overlays are described in [26, 27, 28, 29].



(a) Unstructured overlay. (b) Structured overlay. (c) Hierarchical overlay.

Figure 2.1 – Taxonomy of Peer-to-Peer overlays.

2.1.2 Applications

The P2P communication paradigm we introduced is nowadays harnessed by many distributed applications in several domains. It is mainly used as a building block to find, store and transfer or exchange information in a more or less safe manner. Hereafter, we review some of the main fields of usage.

File sharing

P2P file sharing allows end-users to access any kind of media files. In this context, the peers are end-user computers that are interconnected via Internet. File sharing has been initially popularized with applications such as Napster [30], Bittorent [18], Kazaa [31] and still remains popular and widely used even with the advent of alternative solutions such as streaming, direct download, etc.

Communication and collaboration

Some years ago emails and Internet Relay Chat (IRC) were the most prominent solutions to communicate with others in an asynchronous or live interactive manner.

Today, Voice over IP (VoIP), instant messaging and video chat are supplementing emails and IRC interactions in both enterprise and home networks. Skype[1] is an application that provides these services on top of an hybrid P2P overlay. Even though its internal architecture has changed a bit recently[2], it still remains a P2P network. Thus, many of the 300 millions connected Skype users leverage, unwittingly, the P2P model.

Distributed Computing

Although file sharing is the most well known usage of P2P systems underlined by the medias, especially for copyright infringements, P2P is also a prevalent model for distributed computing. In this field we can distinguish Volunteer Computing (VC) and Cloud Computing (CC). The idea of VC is to give the possibility to any user who disposes of machines and an Internet connection to contribute to projects that require a lot of computing power by donating their unused resources like CPU and storage. The process is really simple and consists of installing an application that turns the hosted computer into a peer that interacts with others based on a P2P model. Many initiatives have emerged the last years based on this VC model. Examples are SETI@home [32], Einstein@home [33] or even Bitcoin [34].

On the other side there is CC where resources are made available with some Quality of Services by cloud providers in exchange for money. Resources are available on-demand. To achieve efficient scalability, cloud platforms such as Amazon EC2, Google Compute Engine or Windows Azure most probably rely on fully decentralized infrastructures similar to P2P systems.

Distributed Storage

Key/Value stores are new systems that are part of the emerging Not only SQL (NoSQL) movement. This class of databases was initially introduced as a shift from traditional SQL databases to enhance read and write performances by providing less guarantees (i.e. not full ACID properties) and simpler query languages. Key/Values datastores are now used by famous companies to store billion of keys

[1] `http://www.skype.com`
[2] `http://goo.gl/dSZu1b`

and terabytes of values in a scalable manner. The underlying structure that is used to scale and achieve these performances in a decentralized environment is a DHT that leverages the P2P model. NoSQL systems are numerous. The most popular are probably Cassandra [35], Dynamo [36] and MongoDB [37]. All are providing good performances but weak consistency. Recently some systems like CATS [38] provide affordable strong consistency and will probably strengthen the NoSQL movement for domains like banking applications where strong consistency is critical.

2.2 Semantic Web

The World Wide Web (WWW) has become an inexhaustible source of information, that grows at an incredible pace and is available to all at any time. This is a fact. However, the WWW as defined at its early stages has several drawbacks. A major issue originates from documents representation that focuses, mainly, on human-readable contents. The information is represented by using markup languages whose the main purpose is to make web documents pleasant to read and navigate to users. Although these documents may contain interesting knowledges, their underlying representation makes information processing to machines really arduous and challenging.

The side effect of this problem is observed when users want for example to perform a search on a specific subject. Search engines crawl Web documents and index information based on keywords. Consequently, they deliver interesting results with pertinent content but mixed up with a lot of irrelevant information. In other words the results returned match the words entered by the users to perform a search but often the context is not captured and a concise answer cannot be deduced. Tim Bernes-Lee assessed the situation and expressed in 1998 the concept of Semantic Web before it defines his vision in [39] as follows:

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

The aims that the Semantic Web tries to achieve is really well captured by

the Figure 2.2. On the left is what browsers see when a simple Web document is interpreted by a machine and on the right is what humans observe. One target of Semantic Web is to fill the gap there is between what machines and humans perceive.

A concrete outcome of increasingly using machine processable formats is that now search engines can provide incredible answers to some natural questions in a concise and relevant manner. Examples are Ask Jeeves[3], the Google Knowledge Graph[4], or even Wolfram Alpha[5]. When you submit a question like "Who was the third president of the USA?", these online services do not only return a list of addresses related to some of the keywords contained in the question but instead, they rely on existing facts extracted and combined from different structured documents to deduce a direct answer along with relevant statements.

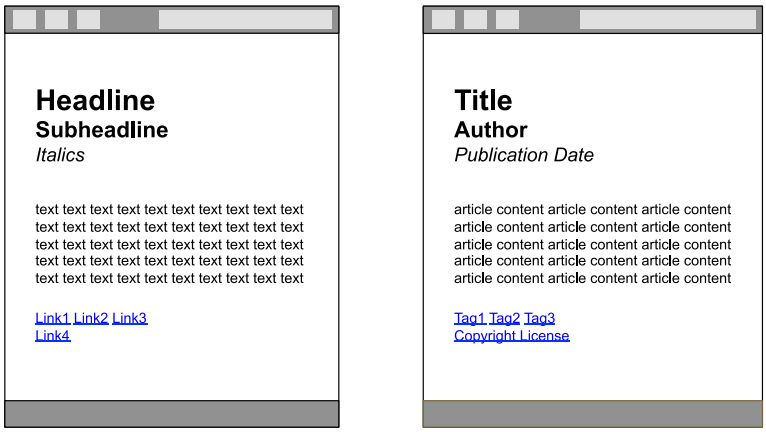


Figure 2.2 – Presentation vs Semantics (taken from [40]).

However, realizing the vision brought by Berners-Lee is not just a matter of using a common structured representation. It requires a full technology stack, referred as the Semantic Web stack to handle different aspects. As depicted by Figure 2.3, it involves several concepts and abstractions whose most are standards and guidelines formulated by the W3C. They are organized in a hierarchical manner and each layer exploits the features and extends the capabilities of the layers below.

[3] `http://ask.com`
[4] `http://google.com`
[5] `http://wolframalpha.com`

The bottom layer refers to well known Web technologies that are Internationalized Resource Identifier (IRI) (a generalization of URI that allows Unicode characters) and Unicode for interlinking, encoding and manipulating text, documents or more generally resources. At the middle, we find standardized technologies like RDF to model information in a machine-processable and machine-understandable manner, SPARQL to retrieve and manipulate data stored in RDF, RDFS that provides basic elements to organize data by sharing vocabularies but also RIF and OWL that enable reasoning over data through rules. Finally, on the top are not yet realized semantic web technologies that relate to logic, trust and thus security aspects.

In the following subsections we will enter into the detail of some technologies, concepts and abstractions we consider important for the remaining of this thesis.

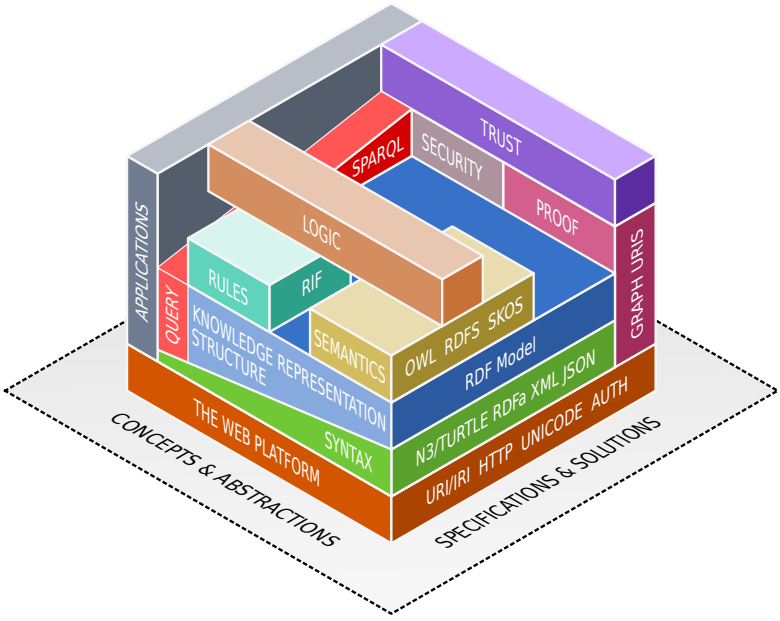


Figure 2.3 – Semantic web stack (originally drawn by Benjamin Nowack).

2.2.1 RDF data model

RDF [41] is a data model standardized by the W3C that aims to encode structured information. It allows to break down any knowledge into independent statements known as triples. A RDF triple is a 3-tuple whose components are respectively

named *subject*, *predicate* and *object*. The subject of a triple denotes the resource the statement is about, the predicate refers to the property of the subject, and the object presents the value of the property.

RDF distinguishes different building blocks as presented below:

- **IRIs** are a complement of URIs that conforms to the syntax defined in RFC 3987 and allows to use Unicode characters. They preserve all the benefits from URIs and thus remain global unique identifiers of resources available on the Web (e.g. `http://www.w3.org/standards/semanticweb`). The main advantage of using IRIs and thus URIs with a resource lies in the fact that anyone can "link to it, refer to it, or retrieve a representation of it" [42]. Consequently, they allow to reason about relationships and ease the integration of distributed information. IRIs are allowed with subjects, predicates and/or objects components of a triple.

- **Literals** are a convenient and intuitive alternative to IRIs for identifying values such as strings, numbers and dates. Literals may be *plain* or *typed*. Plain literals are Unicode strings that are combined with an optional language tag to identify the original tong (e.g. `"привет"@ru`) while typed literals consist of a Unicode string with a datatype URI that determines how the lexical form maps to a literal value (e.g. `"7"^^xs:integer`). Literal values could for example be used to relate objects to their names. Note that literals can only occur as a triple's object.

- **Blank nodes** also dubbed *bnodes* are anonymous resources whose name or identifier is not known or not specified (i.e. no associated URI exists). They may be described as existential variables "simply indicating the existence of a thing, without using, or saying anything about, the name of that thing" [43]. The scope of blank nodes is local to an RDF document. A bnode identifier used in two separate RDF documents can refer to two different things. Blank nodes may be the subject or the object of a triple.

Representation

As mentioned previously, RDF is an abstract model that provides a general method to decompose knowledge into triples and to put them in relation with each other through IRIs. An interesting property that stems from this model, and that reinforces its flexibility, is that triples may be represented into multiple equivalent forms. For example, if we consider the need to describe a book in RDF, a possible manner to model it is the one introduced in Listing 2.1 that rests on a simple representation in the form of a 3-tuples set. First, it describes the book title by referring to the book through its International Standard Book Number (ISBN) URI. Then, more information is added such as the publication date, the publisher name, the publication generic type, but also statements about the book author (i.e. creator) by the intermediate of a blank node used to group creator sub-properties. At this stage we can see that, both, the ISBN and creator elements are shared between multiple triples. This connection between triples lets suppose another possible manner to represent RDF information which is a labeled directed connected graph where nodes are the subjects or the objects of the triples, while edges refer to the predicates of the RDF statements. As depicted by Figure 2.4, this visual representation is a clear explanation of the relationship between RDF building blocks. Note that edges are always oriented from the subjects to the objects elements of triples.

```
(urn:isbn:0201038013, dc:title, "The Art of Computer Programming")
(urn:isbn:0201038013, dc:publisher, "Addison-Wesley")
(urn:isbn:0201038013, dc:creator, _:bnode72)
(urn:isbn:0201038013, rdf:type, dc:BibliographicResource)
(_:bnode72, foaf:firstName, "Donald")
(_:bnode72, foaf:familyName, "Knuth")
(_:bnode72, foaf:homepage, http://www-cs-faculty.stanford.edu/~uno)
(_:bnode72, foaf:pastProject, urn:isbn:0201038013)
```

Listing 2.1 – Book description modeled in RDF as a simple set of triples.

Both representations are abstracts but concrete syntaxes are required to manipulate and exchange triples in real applications. RDF comes will several syntaxes such as Notation3, N-Triples, RDF/XML, etc. Each syntax brings its advantages

and drawbacks. The main differences between them lies in the fact that some are more expressive than others. This may be due to the representation format such as XML that involves naming elements and attributes, but also by the fact that some syntaxes provide a methodology to aggregate redundant information (e.g. namespaces) while others do not.

Named graphs

Carroll et al. [44] bring out in 2005 the need to provide a mechanism for talking about RDF graphs and relations between graphs. For that, they proposed to extend the RDF semantic and existing syntaxes with the concept of named graphs. Concretely, named graphs provide an extra degree of liberty by extending the notion of triples to quadruples (4-tuples). The extra piece of information that is added, named *context* or *graph* value, is an IRI or Blank node placed at the head of each triple and can be used to achieve different purposes. Features that have been suggested are for example the ability to track the provenance of RDF data, to sign RDF graphs or even to provide versioning by capturing snapshots of multiple RDF sources.

Although named graphs are not supported in the current RDF specification since the concept has been proposed after its publication, datastore implementations and query languages such as SPARQL already support and make use of named graphs. In this thesis we consider quadruples, especially to fit the requirements we have regarding the publish/subscribe layer we propose in Chapter 4. However, it is worth to notice that triples could be associated to a default graph and thus behave as quadruples. Reciprocally, a quadruple could be transformed into a triple. It is just a matter of removing one element.

2.2.2 SPARQL query language

Since the writing and publication of the first RDF draft and then the specification, a lot of solutions have been proposed to query RDF knowledges in an efficient and expressive manner. It includes RQL [45], SeRQL [46] and SquishQL [47]. This fragmentation in terms of solutions to query RDF databases called out the Semantic Web community that established with the W3C, a group working towards

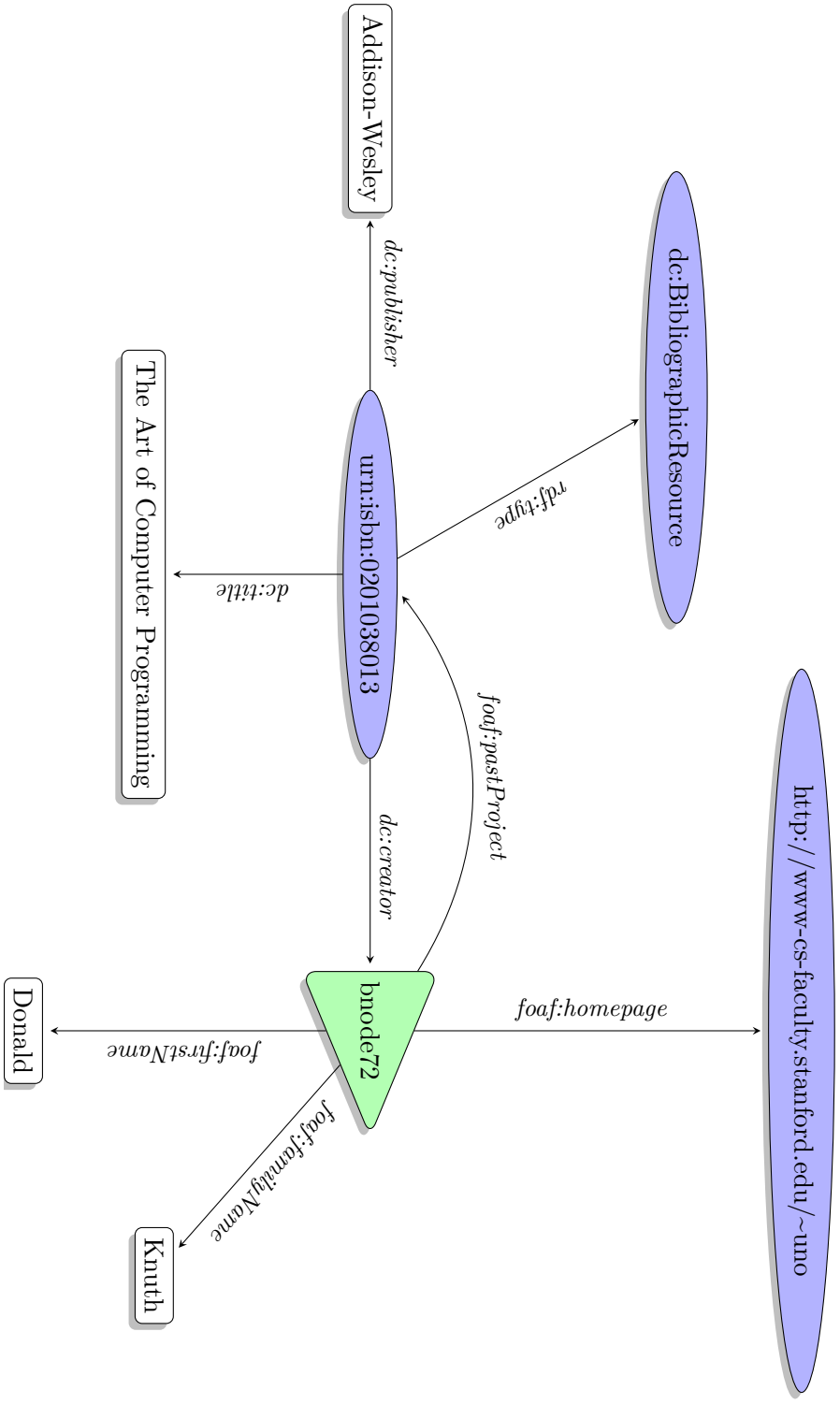


Figure 2.4 – Book description represented as an RDF graph. Oval shapes represent IRIs, rectangles are literals and rounded triangles depict anonymous resources. Labeled edges are predicate IRIs.

a common language taking advantage of the various existing solutions. This realization is called SPARQL Protocol and RDF Query Language (SPARQL).

SPARQL is now a W3C recommendation [48] that plays a very important role in the Semantic Web community. Its foundations are based on the concept of Basic Graph Patterns (BGPs). A BGP is a sequence (conjunction) of triple patterns where each triple pattern is a triple that may contain variables for retrieving unknown values, linking a triple pattern with others, or both. Two triple patterns are linked if they share the same variable. In that case, a join is performed on this variable for values found with each independent triple pattern.

BGPs allow to extract subsets of related nodes in an RDF graph. For example, if we assume there exists a graph database with book definitions modeled as the one introduced in Figure 2.4, then we can retrieve author names of all bibliographic resources by using the SPARQL query presented in Listing 2.2. It consists of one BGP with three triple patterns where the dot character imposes a join between triple patterns that share common variables or the cartesian product[6] between independent triple patterns. Similarly to RDF with the notion of triple that is extended to quadruples with the support of named graphs, triple patterns can be extended to quadruple patterns. In this context, we can say that the query from Listing 2.2 contains three quadruple patterns that share the same graph element (e.g. a default graph value).

```
1 PREFIX dc: <http://purl.org/dc/terms/>
2 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3 SELECT ?authorName WHERE {
4     GRAPH <default> {
5         ?isbn dc:type dc:BibliographicResource .
6         ?isbn dc:creator ?creator .
7         ?creator foaf:familyName ?authorName
8     }
9 }
```

Listing 2.2 – SPARQL query example for retrieving author names of bibliographic resources modeled in RDF.

[6] Assuming two sets $A = \{x, y, z\}$ and $B = \{1, 2, 3\}$, the cartesian product $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.

SPARQL reuses some keywords from SQL such as SELECT, FROM, WHERE, UNION, GROUP BY, HAVING. It also allows for solution modifiers like DISTINCT, ORDER BY, LIMIT and OFFSET that mimic the semantic of those from SQL. However, a main SPARQL characteristic lies in the type of queries supported. SPARQL supports four kinds of queries. ASK queries return a simple boolean to indicate whether a solution exists or not. CONSTRUCT queries create and return RDF graphs in the same manner XQuery [49] builds and returns XML trees. DESCRIBE queries return an RDF graph that describes the resources found. The structure of this graph is not defined in the SPARQL specification and results often differ from an implementation to another. Finally, SELECT queries return a set of variables and their solution in the form of result sets similarly to relational databases. Other characteristics are BGPs building blocks that can be combined with UNIONS, extended with OPTIONALs keywords, and drained with FILTERs that allow to refine results with inequalities or regular expressions.

2.3 The Publish/Subscribe Paradigm

Publish/Subscribe (Pub/Sub) is a messaging pattern that allows users or client applications called subscribers to be kept informed efficiently and gradually about information they are interested in. Unlike one-time or synchronous queries where users formulate meaningful inquiries about their concern and wait for an answer, publish/subscribe systems assume that users register their needs through subscriptions also dubbed continuous queries [50]. As the name suggests, continuous queries are resolved as soon as incoming information or events match subscribers' interests. Here, events act as a means of communication and can be seen as actions or occurrences of something that happened and may point out for example a change or an update in the state of one or more components. Once events are generated, they are published to an event service in charge of performing the matching between the publications and the subscriptions that have been registered. Then, when an event satisfies a subscription that has been previously registered, a notification that reifies the matched event is triggered to the subscriber.

2.3.1 Interaction model

The traditional workflow of interactions between the different entities that are usually involved in a publish/subscribe system is depicted by the Figure 2.5. To summarize publish/subscribe systems are constituted of publishers, subscribers and an event notification service composed of one or more brokers that form a brokering network in charge of mediating events between publishers and subscribers. In this type of systems flow of interactions are unidirectional and asynchronous. They go from subscribers to the event service in order to register subscriptions with the *subscribe* primitive and in two steps from publishers to subscribers through the event service with respectively the *publish* and *notify* primitives. As illustrated by the Figure 2.5, publications produced by publishers are not necessarily forwarded to a subscriber. Indeed, both subscribers may care about different things and the event service may decide to discard some notifications. For instance, *Subscriber 1* is receiving a notification because the published event is satisfying its interests. However, *Subscriber 2* does not receive any notification since the subscription registered in the event service is not satisfied by the event coming from the publisher.

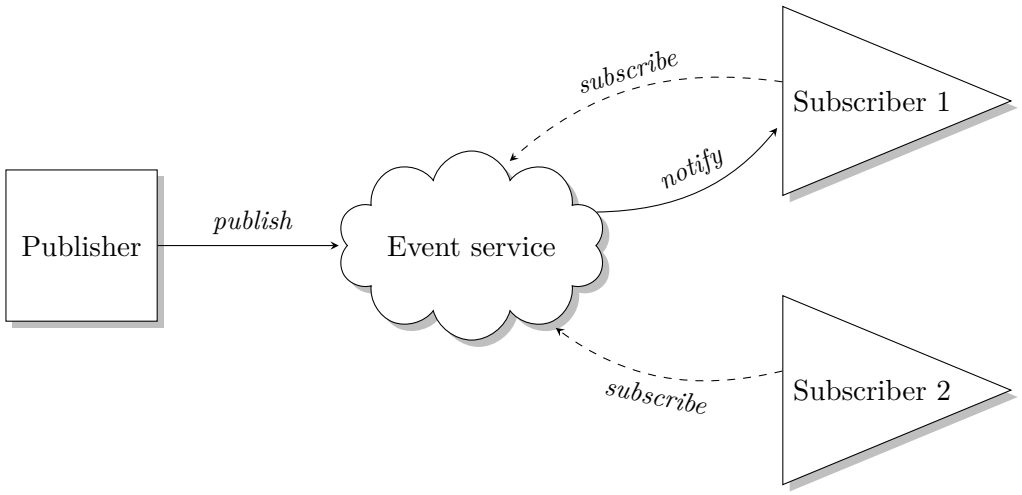


Figure 2.5 – Publish/Subscribe interactions.

Some publish/subscribe systems also rely on an additional primitive dubbed *advertise* to publish advertisements. Advertisements are used to inform the system about the kind of information publishers are willing to send. Their aim is to reduce the overall information flow and thus to save time and effort in disseminating

events. However, the use of advertisements is beneficial when publishers exhibit stable publishing patterns only.

2.3.2 Characteristics

The last years publish/subscribe systems have demonstrated their flexibility, modularity and responsiveness by being used with success in a broad set of application scenarios that ranges from information dissemination, network monitoring, ubiquitous systems or even mobile systems. The main characteristic behind the strength of the publish/subscribe communication style lies in its decoupling in terms of space, time and synchronization between publishers and subscribers.

- **Space decoupling** refers to references, i.e. the identity and the location of subscribers is not known by the publishers and reciprocally publishers do not hold any reference to subscribers. They are mutually unaware of each others. This is made possible by means of the event service that acts as a mediator. The space decoupling characteristic is very interesting in highly dynamic environments where participants are frequently leaving or joining the system.

- **Time decoupling** refers to the eventuality that publishers publish events even though no subscriber is present, or reciprocally, subscribers register a continuous-query when no publisher is contributing. In this way, if an unexpected outage occurs for instance on one or more publishers, subscribers are not strictly affected since they can still receive event notifications by means of the event service.

- **Synchronization decoupling** refers to the process of triggering a publish or subscribe operation without waiting for an acknowledgment or a response. Communications are asynchronous. This is really analogous to how people proceed when they exchange by mails. Once an email is sent we don't expect to receive an answer immediately and we don't even know whether a reply will come back or not later. The transmission is said one-way. In that case, we can start to do some other work while the message is in transit. In publish/subscribe, the behavior is the same. Publish and subscribe operations

executed by clients return immediately, thus the clients can continue their standard flow of execution even if the operation is not yet completely handled by the event service. Consequently, a strong benefit is that operations can implicitly overlap and thus increase parallelism.

The main advantage that lies in the fact that entities are loosely coupled is also the major drawback of publish/subscribe systems. The mediation layer might, under unexpected events such as power failure, network partition, etc. not trigger notifications and when this situation occurs there is no way to know whether the delivery has succeeded or failed. In that case, tighter coupling or strong guarantees on the event service by providing for example replication is required.

2.3.3 Filtering mechanisms

Publish/Subscribe systems introduced the last two decades differ from an user point of view by the expressivity of the subscription language and consequently the filtering mechanisms that are employed for selecting events to forward and deliver. According to this aspect, we review hereafter the main event-based filtering mechanisms.

Channels-based filtering

The first and concrete representation of the publish/subscribe paradigm is based on the concept of channels [51] where events are produced and forwarded to named channels. Then, interested parties may consume all events crossing over a particular channel by pointing it with its name similar to a keyword. This scheme is the one implemented by many forerunner systems.

Topic-based filtering

Topic/subject based filtering can be seen as an extension of the simple channel approach. In the subject-based model, publishers annotate every event they generate with a string denoting a distinct topic. Generally, this string is expressed as a rooted path, similar to an URI, in a tree of subjects [52]. For instance, an online research literature database application (such as IEEE Xplore or Springer

Archives) could publish events of newly available articles from the Semantic Web research area under the topic */Articles/Computer Science/Proceedings/Web/Semantic Web/*. This kind of topic will then be used by subscribers which will, upon subscription's generation, explicitly specify the topic they are interested in with optionally a wildcard to perform pattern matching on subject names. Based on this subscription they will receive all related events. The topic-based model is at the core of several systems such as Scribe [53] and Vitis [54]. A main limitation of this model lies in the fact that a subscriber could be interested only in a subset of events associated to a given topic instead of all events. In other words, the tree-based classification severely constrains the expressiveness of the model as it restricts events to be organized using a single path in the tree. Some inner re-organizations are possible. A solution could consist of associating an event to several hierarchical topics but it will lead to several issues that range from duplicate publications and notifications to a growing number of information to exchange.

Content-based filtering

Content-based filtering provides a fine-grained approach that allows the evaluation of filters on the whole content of the events that are published. In other words, it is the data model and the applied predicates that exclusively determine the expressiveness of the filters. Subscribers may express their interests by specifying predicates over content of events they want to receive. These constraints can be more or less complex depending on the subscriptions types and operators that are offered by the subscription language. Available subscription predicates range from simple comparisons, conjunctions, disjunctions to regular expressions or even XPath expressions on XML events. Content-based filtering is now the most general scheme supported by recent and famous publish/subscribe systems such as Hermes [55] or JEDI [56].

Type-based filtering

On the one hand topics tends to regroup events that present similar properties in terms of content or in structure. However, this classification is not mandatory and may be made based on predefined external criteria. On the other hand,

with content-based filtering events are classified based on the content of events. Consequently, to achieve the same functionality as that of topic based systems, subscribers have to filter out irrelevant events. Type-based filtering has been proposed to sweep away inappropriate events according to their type [57]. The idea is to enforce a natural and inherent subscription scheme that avoids to classify events explicitly through topics with the implicit desire to complement content-based filtering by acting as an efficient colander to prefilter events.

2.4 ProActive Middleware

ProActive is a Java middleware, historically introduced in [58], that provides the programming and runtime facilities to build and deploy parallel, distributed and concurrent applications. It is built on top of standard Java APIs, namely the RMI and introspection API. Consequently, ProActive applications can run on any operating system that disposes of a compatible virtual machine.

Today, ProActive became a mature middleware that comes with multiple built-in features. Figure 2.6 summarizes the main aspects the library can deal with. The top level layer brings out the different programming models it supports (Branch & Bound, Components, Groups, and Master-Slave). These programming models are built upon the concept of objects that are said active because they dispose of their own thread of control. With active objects every method invocation is handled by the object's thread of control. More details regarding the active object model are given in the next subsection.

The middle layer summarizes the various services the ProActive middleware features. It includes a fault tolerance mechanism based on a checkpointing protocol [59], a method to wrap legacy code in order to control and interact for instance with MPI applications [60], but also the capability to migrate active objects [61], a security framework for communications between remote active objects [62], and many more.

At the bottom layer, we find infrastructure and communication related features. It contains a deployment framework that allows to deploy active objects on multiple different infrastructures without changes in the application source code [63], a resource/scheduler manager for dynamic provisioning and scheduling of ProAc-

tive applications in cloud environments [64], and a graphical environment which enables programmers to interactively control and debug the distribution aspect of ProActive applications [65]. Furthermore, the library allows active objects to communicate remotely by using several network protocols such as RMI, HTTP, or even custom ones according to required speed, security, error detection, firewall or NAT friendness properties. It is also possible to export active objects as Web services and to invoke methods with the standard SOAP protocol.

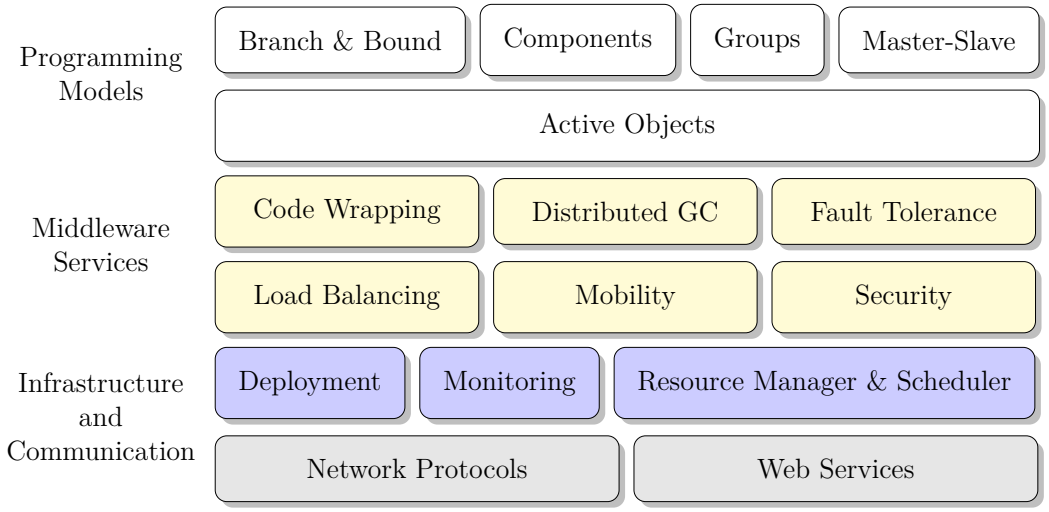


Figure 2.6 – ProActive middleware features.

Below, we review some important notions about ProActive that are extensively used for implementing the middleware developed within this thesis.

2.4.1 Active objects

The main features of the library are based on the active object model. Active Objects (AOs) are medium grained entities whose method invocation is decoupled from method execution with the help of the Meta Object Protocol (MOP) pattern. That way, developers concentrate only on the business logic of applications they develop since the distribution of active objects is transparent. Indeed, invoking methods on remote active objects is similar to invoking a method on a standard Java object by using the dot notation. Neither additional piece of code to establish connections between remote entities nor specific class to extend or interface to

implement, like with RMI, is required.

Figure 2.7 depicts the meta object architecture for a typical active object. On one hand there is the active object instance that is built over an existing Java object *MyObject* by extending it transparently at runtime with a Body object that acts as an intermediate to hide network communications from a user point of view. The Body relies on different feature meta objects to receive method calls (also named requests once received by the Body), to execute requests, to send optionally a reply to the caller, but also some others to handle migration, fault tolerance, etc. On the other hand, an active object is always indirectly referenced through a proxy and a stub which is in our case a subtype of the object *MyObject* which is either pre-compiled or generated at runtime.

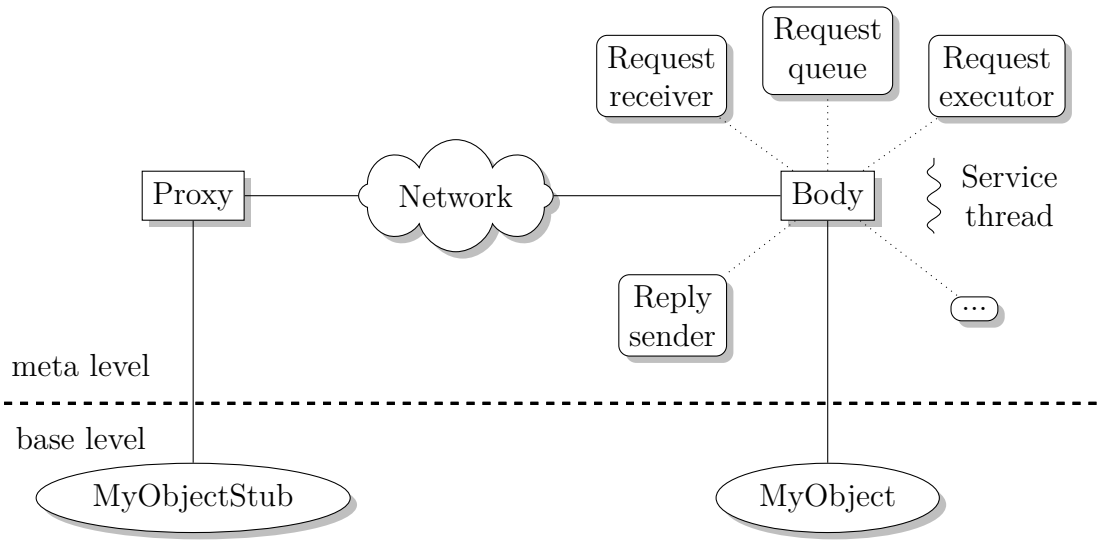


Figure 2.7 – Meta object architecture.

When a user performs an invocation to the active object, this is in fact an invocation on the stub object *MyObjectStub*. The stub is in charge to create a reified representation of the invocation by building an object representation of the method call along with its parameters before passing it to the proxy. Then, the proxy transfers the reified invocation (i.e. the method call object) as a message to the request receiver of the Body object, possibly through the network. Afterwards, the method call is queued in a request queue. Later, one request is picked from the request queue by the request executor according to a desired serving policy. Notice

that with the standard active object model only one request can be executed at once and the default policy is First In, First Out. As a consequence, data races are prevented without using synchronized blocks. Finally, if a response is associated to the method call that has been executed, it is returned to the caller by means of the reply sender.

Semantic of communications

Seeing that all communications made with ProActive are through method invocations, the communication's semantics depends upon the method's signature. Unfortunately, the resulting invocations may not always be asynchronous. This constraint is caused by the MOP pattern that has some limitations related to the serialization of parameters and invocation results. Consequently, two communication idioms regarding invocations are distinguished:

- **Synchronous invocations**
  With a synchronous method call, the caller thread is blocked until the method completes its execution and optionally returns a result. In ProActive, a method invocation is synchronous if the considered active object method's signature declares to return a non reifiable object or to throw an exception. An object is reifiable if its associated class is declared not final, serializable and embeds a public constructor. Primitive data type values (i.e. boolean, char, int, etc.) are not reifiable.

- **Asynchronous invocations**
  Unlike synchronous invocations, asynchronous method calls allow the invoker to continue its standard flow of execution even if the method has not completed. Since ProActive introduces futures as placeholders for results, two subtypes of asynchronous invocations are distinguished:

  - **One-way asynchronous invocation**
    If the method does not throw any exception and does not return any result, the invocation is one-way. The invocation is asynchronous and the process flow of the caller continues once the reified invocation has been received by the active object.

– **Asynchronous invocation with futures**
Methods that do not thrown any exception and declare a reifiable type as result have their invocations implicitly handled with futures. In the same way that one-way asynchronous calls, asynchronous invocations with futures incur a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee, in order to ensure a causal dependency. However, futures allow the process flow of the invoker to continue once the reified invocation has been received by the active object, this even if a result is expected. The synchronization is data based and handled by a mechanism known as wait-by-necessity [66]. In other words, when the caller tries to access a future, the caller thread is blocked until to receive the associated value. Since the caller thread is in a waiting state, it cannot continue with the service of the next request.

A future may be passed as parameter of a method call, even if it is not yet resolved. Thus, nesting method calls that return futures does not trigger a wait-by-necessity. Once a future is used all disseminated references are updated by means of a mechanism called automatic continuations [67].

Even though active objects are mono-threaded, which avoids data races without the use of synchronized blocks, users must care about deadlocks. Consider a simple scenario with a peer that routes a request with message passing emulated on top of remote method invocations. In that case, when the request passes again through the sender, with a re-entrant call, a deadlock may occur if the peer is already waiting on itself for a response. A solution to address this issue consists in modifying the application logic but it is often difficult to program and it incurs several drawbacks. Another alternative is to rely on Immediate Services (ISs)[7]. However, methods declared as IS are handled in their own thread but in a synchronous manner regarding the caller, whatever the method signature. Moreover, even if parallelism is improved, users must be careful about part of code to synchronize explicitly with locks to prevent race conditions. Last but not least, futures

[7] `http://goo.gl/m2oIbn`

are no longer usable and their benefit is lost.

Multi-active objects have been proposed recently as an elegant manner to address this issue and enhance efficiency on multicore machines. This extension is introduced in the next subsection.

2.4.2 Multi-active objects

The Multi-Active Object (MAO) model [68] is an extension that overcomes the limitations of the founding model, that is handling requests sequentially, by allowing method calls to be multithreaded. Its concept relies on method annotations to decide which requests can be run in parallel with others through the definition of compatibility groups.

Groups are declared with the `@Group` annotation and the assignation of methods to groups is done with the help of the `@MemberOf` annotation. Compatibilities are defined between two groups through the `@Compatible` annotation. The idea behind compatibilities is that two groups that are compatibles may have their methods executed simultaneously. By default, groups whose compatibility is not set are assumed conflicting with others and thus incompatible. Usually, two groups have to be set compatible if their methods do not access the same data or if the scheduling order and the concurrent accesses on the same resource are protected by the programmer by means of locks or synchronized blocks.

Listing 2.3 gives an illustrative example on how to define multi-active groups and compatibilities for a peer class exposing four methods, each having different purposes. Methods *leave* and *join* belong to a *structure* group and are used to manage the overlay structure. Declaring these methods compatible would imply that the programmer implicitly synchronizes the accesses to the common resources since both require an exclusive access. In the same manner, joining nodes and routing at the same time from a same peer is not recommended. Owing to the conflicting nature of these operations, no compatibility is set. However, routing and retrieving monitoring information are two concepts that access disjoint resources and can be handled in parallel in the same peer. This is why a compatible annotation entry is declared. In addition, an optional `selfCompatible` attribute parameter can be specified with the definition of a group to indicate whether requests from a same

group can be executed concurrently or not. In the example which has been introduced, *monitoring* and *routing* groups are declared compatible and self compatible given that associated methods should only read variable values.

```
1  @DefineGroups({
2      @Group(name = "structure", selfCompatible = false),
3      @Group(name = "routing", selfCompatible = true),
4      @Group(name = "monitoring", selfCompatible = true)})
5  @DefineRules({
6      @Compatible({"routing", "monitoring"})})
7  public class PeerImpl {
8      @MemberOf("structure")
9      public JoinResponse join(Peer landmarkPeer) { ... }
10
11     @MemberOf("structure")
12     public void leave() { ... }
13
14     @MemberOf("routing")
15     public Response execute(Query query) { ... }
16
17     @MemberOf("monitoring")
18     public LoadInformation getLoad() { ... }
19 }
```

Listing 2.3 – Groups and compatibility definition using multi-active objects annotations for a class that embodies a peer instantiated as an active object.

Furthermore, multi-active objects provide a mean to decide about compatibility at runtime. To do so, programmers can indicate an expression or a compatibility function to evaluate with the help of a `condition` parameter to add along with the `@Compatible` and/or `@Group` annotation. In that case an optional group parameter is also required for the groups that are involved since the compatibility between two requests is decided as a function depending on three parameters: the group parameter of the two requests, and the status of the active object.

Concretely, if we refer to Figure 2.7, multi-active objects are implemented by means of a new request executor that allows multiple threads to be spawned and one or more request(s) to be picked from the request queue in order to be

served. Briefly speaking, a request is selected, removed from the request queue and served if it is compatible with other requests that are running and all requests that are before it in the request queue. This default and configurable serving policy, called *First Compatible First Out*, maximizes parallelism while ensuring that two incompatible methods cannot be run in parallel.

Although multiple threads are used to enhance efficiency on multicore machines, managing too many threads can be harmful due to memory consumption explosion or too much concurrency with regards to the number of cores available on the machine where the active object is deployed. For this reason, the implementation of multi-active objects come also with an API to limit the maximum number of threads used by an active object. It is possible to set either a strict limit on the maximum number of threads to be used, or to limit only the maximum number of threads that are active and running but not those that are in a waiting state (e.g. the threads that are waiting for a future). The former thread management policy is called *hard limit* and may obviously lead to deadlocks whereas the latter called *soft limit* prevent deadlocks with re-entrant method calls.

The multi-active objects library is at the heart of the middleware we propose within the context of this thesis. We will see in Chapter 6 that the various parameters to use with multi-active objects in our implementation have been empirically tested. Also, we will highlight some minor features we have added to the library in order to resolve some issues that may arise when it is used in complex scenarios.

2.4.3 Components

Component oriented programming provides many facilities for the development of complex and robust applications. The building blocks are components which are large grain software entities or modules offering predefined services. From a user point of view, components act as black boxes that communicate with others via provided and required interfaces. Therefore, they can be reused and composed by third parties without any knowledge of their internal workings.

ProActive provides an implementation of Grid Component Model (GCM) [69] based on its active object model. GCM is an extension of the popular Fractal component model [70]. It aims to ease the programming of distributed applica-

tions by targeting their design, deployment, reusability and efficiency. As Fractal, GCM/ProActive allows for hierarchical composition, separation of functional and non-functional concerns but also considers autonomic composition [71], collective communications [72] and deployment.

Since GCM/ProActive inherits the hierarchical property of Fractal components, a GCM component may exhibit a recursive structure. Thus, a component is either a composite (i.e. composed of other components) or a primitive (i.e. a single component that encapsulates a basic functional block). Figure 2.8 depicts a composite component with two primitive subcomponents. Provided interfaces to be consumed by clients, also dubbed server interfaces, are represented on the left hand side of a component box. Interfaces drawn on the right hand side depict required interfaces, or client interfaces, that are consumed by other components through bindings that carry messages. On the top of a component box we find controllers that are interfaces used to externalize non-functional aspects. Their purpose is to manage component properties and assembly (e.g. configuration, dynamic reconfiguration, monitoring, security, etc.) through passive Java objects encapsulated in the membrane of a component.

The definition of components and their bindings can be made programmatically or through Fractal ADL files in order to be independent from the implementation, thus requiring no recompilation. It is also worth to notice that GCM enables components to be spread over different machines in an easy and transparent manner by means of XML descriptors.

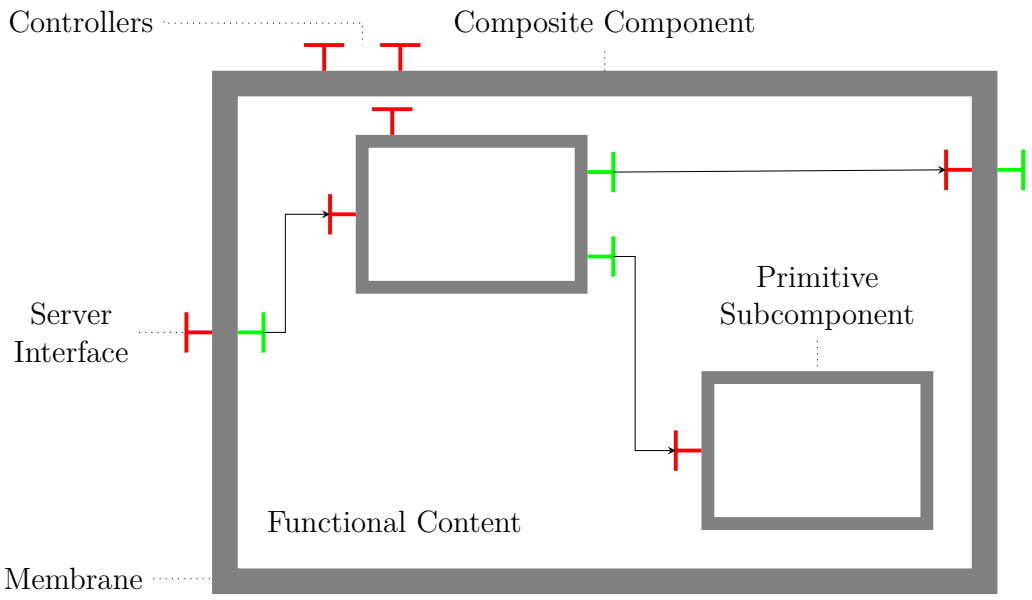


Figure 2.8 – Standard Fractal/GCM component.

Chapter 3

Distributed RDF Storage

Contents

In this chapter we present our first contribution that relates to the implementation of a distributed RDF storage infrastructure. The infrastructure that is described was originally proposed by Imen Filali in her thesis [73]. This chapter starts with a brief related works section about decentralized systems for storing and retrieving RDF data. Then, we introduce the popular CAN P2P protocol which is the underlying P2P overlay network we rely on for routing messages and, indirectly, for achieving scalability. Afterwards, we motivate and discuss the design choices and the adjustments we made regarding the CAN protocol before

explaining how messages are routed with our modifications. In a penultimate section we describe how RDF data is indexed in the P2P network and how SPARQL queries are executed. Finally, we provide the results we got by experimenting our solution on the Grid'5000 testbed and we conclude with a summary and some perspectives.

3.1 Related Works

Although the Semantic Web movement is still relatively new, many research efforts have already been made by companies and academies to improve how RDF data, and especially how the underlying tuples are stored and then retrieved with SPARQL. In this section we give an overview about different strategies and solutions that have been proposed in both centralized and distributed environments.

3.1.1 Centralized RDF stores

Many different representations have been used over the last years to store RDF data in a centralized environment. This is due to the fact that RDF is a data model that does not impose any storage organization. In other words, it is a method to express any fact in a structured manner but anyone can write down triples in multiple different ways that still preserve the original information and structure. However, it is important to notice that storage representations are strongly affected by the SPARQL query language involved in the expression of queries. In SPARQL, building blocks are BGPs made of triple patterns. Thus, the critical challenge RDF engines try to achieve is the efficient resolution of BGPs. BGPs are usually processed in two phases known as *scan* and *join*. First, BGPs are extracted from a SPARQL query and decomposed into a set of triple patterns. Then, one or more tables are scanned in order to extract intermediate results. Finally, since triple patterns may share common variables, the resulting intermediate values have to be joined on these variables to produce the final answer. In this context, the different strategies adopted aim to reduce the time required to execute the two phases by minimizing for instance the number of join operations to perform.

In its early stages, RDF stores were storing RDF data as a set of tuples in

relational databases. This includes for instance 3store [74], Jena [75], Oracle RDF Match [76], RDFSuite [77] and Sesame [78]. The main state of the art strategies are described in [79]. They may briefly be summarized as follows:

- **Tuple tables** are a natural approach to store RDF data in relational databases. The idea is to put tuples (triples or quadruples) in a 3 or 4 columns table. In this manner, each row represents an RDF statement. However, this representation is inefficient because queries that embed for instance k quadruple patterns with common variables require $k-1$ self-joins over this very long table. Thus, its leads to many disk accesses as it is impossible to cache the entire table in memory.

- **Property tables** is another strategy that consists of building one or more tables according to common set of attributes that occur frequently. Thus, if subject elements are frequently associated with a common set of predicate properties such as *rdf:type*, *dc:title* and *dc:create*, then a table with the subject as a key and the other attributes as the following columns is created. As a consequence, queries that have multiple triple or quadruple patterns that share a same subject variable are now join free. They may be resolved with a simple scan over a predicate table as long as all attributes of a query are covered by a single predicate table. Nevertheless, this technique has some drawbacks. Heterogeneous records are not supported. Thus, since not all subjects share a common set of attributes, some entries may require NULL values. Moreover, multi-valued attributes are problematic. Especially, if a subject has more than one object value for a given property (e.g. a book can be written by two people), then one or more object values are duplicated. Additionally, a query that looks at multiple property tables may require complex joins with intermediate results.

- **Vertical partitioning** entails properties' based tables creation. One two columns table is created for each unique property (predicate). The first column contains the subjects of all tuples that share the table predicate whereas the second column contains the associated object values. Unlike property tables, vertical partitioning support multi-valued attributes and

heterogeneous records. Besides, joins give interesting performance results because the tables are smaller than other strategies.

Solution layered above Relational Database Management Systems (RDBMs) have quickly thrown doubts about their scalability with regards to the increasingly growing set of structured information to handle. Indeed, relational DBMs have been designed long time ago when hardware characteristics were much different than today and where deployment was targeting a single machine. Stonebraker *et al.* argue and demonstrate in [80] that major RDBMs solutions can be outperformed by at least 1 or 2 orders of magnitude with specialized engines targeting specific applications requirements. Almost at the same period, Abadi *et al.* [79] showed that vertically partitioning applied on a column-store outperforms an RDF store implementation on a row-store engine (i.e. traditional relational database), that uses tuple tables, with a factor 32. More recently, Sidirourgos *et al.* [81] performed an independent assessment of the results and confirmed the performance trends. Since, many systems have been reworked to provide solutions to store and query RDF documents more efficiently. These systems that no longer rely on RDBMs are called native RDF stores.

The idea behind native RDF stores consists of creating custom partitioning and indexes, from time to time similar to the strategies introduced previously with relational databases. However, the main difference lies in the fact that specific low level optimizations are possible. For example, Jena TDB [82] is a native persistent storage engine whose indexes are implemented using conventional B+Trees with additional forward linking of the leave blocks to ease scans. B+Trees are custom implemented and they support only what is necessary for the purpose of indexes, thus they strip out a lot of the overhead (no row overhead: no null map, no per-row locking, etc.). Among others, RDF3X [83] adopts a strategy similar to tuple tables but addresses the issue about expensive self joins by creating an interesting set of indexes. Tuples are sorted lexicographically in a compressed B+Tree, which allows the resolution of BGPs with range scans. Also, similarly to many RDF stores, RDF3X replaces all literals by fixed size identifiers using mapping dictionaries. This approach has two benefits. Frequently occurring values are stored once and identifiers are compared must faster for equality than long strings. More

recently Yuan *et al.* proposed TripleBit [84]. Similarly to others, it uses mapping dictionaries to improve scan and joins phases. However, unlike others, it is based on a triple matrix storage structure that features compression for storing large RDF graphs more efficiently.

3.1.2 Distributed RDF stores

Although centralized RDF stores are sometimes enough in production for small workloads, they suffer from the traditional limitations of centralized approaches, namely a single point of failure, performance bottlenecks, etc. As an alternative, fully decentralized and distributed systems have been proposed to overcome some of these limitations. This section presents a selected sample of research works that support the storage of RDF data and the execution of SPARQL queries in a distributed environment. As we will see, the solutions differ by their underlying data storage facilities. Some are built as a pure P2P system (where there is no centralized authority) or on top of a P2P overlay network whereas some others rely on a NoSQL store. Regarding pure P2P solutions, we consider only those that make use of a structured overlay. Others based on unstructured overlays such as Edutella [85], Bibster [86] and S-RDF [87] are deliberately left aside since they require to flood the whole network or to use heuristics to lookup a resource, which leads to scalability issues or no solution for unpopular resources. Also, solutions based on a Master-Worker model such as OWLIM Enterprise [88] are out of the scope of this related work section because critical operations such as write requests have all to pass through a single node.

RDFPeers

RDFPeers [89] is the first P2P system that came up with the idea to use DHTs in order to implement a distributed RDF repository. It is built atop Multiple Attribute Addressable Network (MAAN) [90] which is an extension of Chord [21]. Similarly to Chord, nodes are virtually organized into a ring. Each node owns an identifier that represents its position in a circular identifier space of size N, and has direct references to its predecessor and successor in the ring. A successor of a node n is defined as the first node that succeeds n along the clockwise direction in

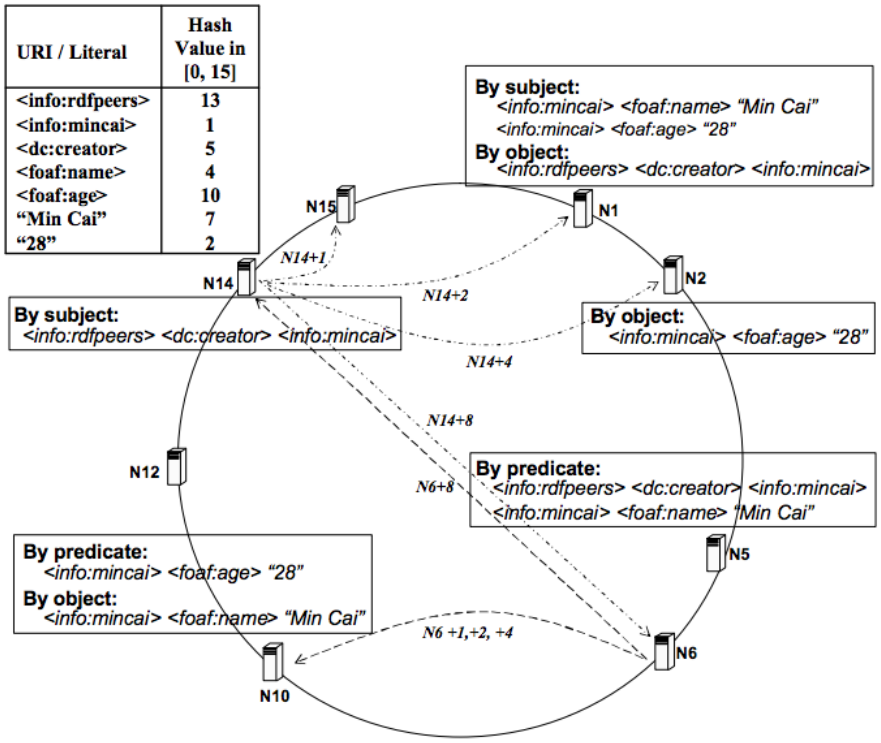


Figure 3.1 – RDF data storage in an RDFPeers network (taken from [89]).

the ring space. Additionally, a node keeps $m = \log N$ routing entries, called fingers. Fingers are successors in power of two. Finger i is equal to successor $(n + 2^{i-1})$ mod N with $1 \leq i \leq m$. Figure 3.1 shows a RDFPeers network with 8 nodes in a 4 bits identifier space. It also depicts the fingers of nodes $N6$ and $N14$ drawn with dashed arrows originated from them. Data indexation and query processing mechanisms are described below.

- **Data indexation**
  RDFPeers considers 3-tuples only. Each RDF term is used as a DHT key at the MAAN level. More precisely, a RDF triple labeled $t = (s, p, o)$ is indexed three times by applying a hash function on the subject, the predicate and the object. Then, the triple t is stored on peers responsible for these hash values. For better understanding consider the storage of the following three triples inside the RDFPeers infrastructure: `(info:rdfpeers, dc:creator, info:mincai)`, `(info:mincai, foaf:name, "Min Cai")` and `(info:mincai,`

`foaf:age, "28"^^xsd:integer)`. Figure 3.1 depicts the hash values for all the IRIs and literal elements of these triples. Suppose now that node $N6$ receives a store message that aims to insert the triple `(info:mincai, foaf:age, "28"^^xsd:integer)` in the network according to the hashing of the predicate element (i.e., *foaf:age*). Given that *hash(foaf:age)=10*, $N6$ will route the triple to $N10$, based on its fingers.

- **Query processing**
  RDFPeers supports three kinds of queries that are summarized below.

  - **Atomic triple pattern queries** are queries where the authors suppose there is always at least one constant value. For instance, a query like $(s, ?p, ?o)$ will be forwarded to the node responsible for $hash(s)$. All atomic queries take $O(\log N)$ routing hops to be resolved except queries in the form of $(?s, ?p, ?o)$ which require $O(N)$ hops in a network of N peers.
  - **Disjunctive and range queries** are a type of query optimized by RDFPeers through the use of a locality preserving hash function[1]. Indeed, when the domain of a variable is limited to a range, the query routing process starts from the node responsible for the lower bound. It is then forwarded linearly until received by the peer responsible for the upper bound. In the case of disjunctive range query like $(s, p, ?o)$, $?o \in \cup_{i=1}^{n}[l_i, u_i]$ where several ranges have to be satisfied, intervals are sorted in ascending order. The query is forwarded from one node to the other, until it is received by the peer responsible for the upper bound of last range. Disjunctive exact queries such as $(s, p, ?o)$, $?o \in \{v_1, v_2\}$ are resolved using the previous algorithm since they are considered as a special case of disjunctive range queries where the lower and the upper bounds are equal to the exact match value.
  - **Conjunctive queries** are supported by RDFPeers as long as they are expressed as a conjunction of atomic triples patterns or disjunctive

[1]Locality preserving hash functions are hash functions where the relative distance between input values is conserved in output values.

range queries for the *same* subject. Constraints' list can be related to predicates or/and objects. To resolve such type of query, authors use a *multi-predicate query resolution* algorithm. This algorithm starts by recursively looking for all candidate subjects on each predicate and intersects them at each step before sending back final results to the query originator.

Although RDFPeers supports several kinds of queries, it has a set of limitations especially in the query resolution phase. This includes the attribute selectivity and the restrictions made at the level of supported queries. The attribute selectivity is related to the choice of the first triple pattern to be resolved. Low selectivity of an attribute leads to a longer computational time to manage the local search as well as a greater bandwidth consumption to fetch results from one node to the other, because many triples will satisfy that constraint. As an example, the predicate *rdf:type* seems to be less selective, as it can be more frequently used in RDF triples than others (e.g. predicates that support range queries). Despite the attribute selectivity parameter having an important impact on the performance of the query resolution algorithm, RDFPeers does not provide a way to estimate such a parameter. Another issue is related to conjunctive triple pattern queries which are not fully supported and are restricted to conjunctive queries with the same subject. Therefore, it does not support arbitrary joins.

RDFCube

Monato *et al.* propose RDFCube [91], an indexation scheme built along together with RDFPeers. Their solution, that is based on a three-dimensional CAN [20] like coordinate space, improves the execution of SPARQL queries compared to RDFPeers by introducing an index that allows to eliminate some peers that are not involved in the matching. To achieve this purpose, their 3-dimensional coordinate space is made of a set of cubes that have the same size and that are called *cells*. Each cell contains an *existence-flag*, labeled *e-flag*, indicating the presence (*e-flag=1*) or the absence (*e-flag=0*) of triples in that cell. The set of consecutive cells that belong to a line parallel to a given axis forms a *cell sequence*. Cells belonging to the same plane perpendicular to an axis form the *cell matrix*.

- **Data indexation**
  Once an RDF triple *t=(s,p,o)* is received, a flag is mapped to the cell of the RDFCube where the point *p=(hash(s), hash(p), hash(o))* belongs to. Additionally, the triple is inserted in the running RDFPeers instance.

- **Query processing**
  As for RDF triples, a query is also mapped into a cell or a plane of RDFCube based on the hash values of its constant part(s). As a consequence, the set of cells including the line or the plane where the query is mapped are the candidate cells containing the desired answers. Note that RDFCube does not store RDF triples, however, it stores bit information of e-flags. Therefore, the interaction between RDFCube and RDFPeers is as follows. On one hand, RDFCube is used to store *(cell matrixID, bit matrix)* pairs such as the *matrixID* which is a matrix identifier and represents the key in the DHT terminology, while *bit matrix* is its associated value. On the other hand, RDFPeers stores the triples associated with the bit matrix information. This bit information is basically used to speed up *join query* processing by performing an *AND* operation between bits and transferring only the relevant triples. As a result, this scheme reduces the amount of data that has to be transferred between nodes.

Battré et al.

In [92], Battré *et al.* propose a data management strategy for DHT based RDF stores. As many others, the proposed approach indexes a RDF triple by hashing its subject, predicate and object. It is worth notice that the proposed solution takes into consideration RDFS reasoning on top of DHTs by applying RDFS reasoning rules.

- **Data indexation**
  The main difference compared to other RDF based structured P2P approaches is that nodes host different RDF repositories in order to make a distinction between local and incoming knowledge.

  – **Local triples repository** stores triples that originate from each node.

Local triples are disseminated in the network by calculating their hash values based on subject, predicate and object terms before being sent to nodes responsible for the appropriate parts of the DHT key space.

- **Received triples repository** simply stores the incoming triples sent by other nodes.
- **Replica repository** ensures triple availability under high peer churn. The node with an identifier numerically closest to the hash value of a triple becomes root node of the replica set. This node is responsible for sending all triples in its received database to the replica nodes.
- **Generated triples** repository stores triples that are originated from applying forward chaining rules on the received triples repository, and they are then disseminated as local triples to the target nodes. This repository is used for RDFS reasoning.

In order to keep the content of the received triples repository up to date, especially under node leaving or crashing, triples are associated with an expiration date. Therefore, the peer responsible of that triple is in charge of continuously sending update messages. If the triple expiration time is not refreshed by the triple owner, it will be eventually removed from these repositories. This approach takes care of load balancing issues specially for uneven key distribution. For instance, the DHT may store many triples with the same predicate *rdf:type*. As subject, predicate and object will be hashed, the node responsible for the *hash(rdf:type)* is a target of a high load. Such situation is managed by building an overlay tree over the DHT in order to balance the overloaded nodes.

- **Query processing**
  In another work [93], one of the authors proposes a query algorithm with optimizations based on a look-ahead technique and Bloom filters [94]. Knowledge and queries are respectively represented as *model* and *query* directed graphs. The query processing algorithm basically performs a matching between the query graph and the model graph. On one side, there is the candidate set which contains all existing triples, and on the other side, there is a candi-

date set containing the variables. These two sets are mutually dependent, therefore a refinement procedure has to be performed to retrieve results for a query. This refinement proceeds in two steps. The first step starts from the variable's candidate set. A comparison is done with the candidate sets for each triple where the variable occurs. If a candidate does not occur within the triple candidate set, it has to be removed from the variable candidate set. The second step goes the other way around, that is, it looks at the candidate set for all the triples and removes all candidates where there is a value not matching within the variable's candidate set.

The look-ahead optimization aims at finding better paths through the query graph by taking into account result set sizes per lookup instead of the number of lookups. This yields fewer candidates to transfer but the tradeoff is that it incurs more lookups. The other optimization, using Bloom filters, considers candidates for a triple (x, v_2, v_3), where x is a fixed value and v_2 and v_3 are variables. When retrieving the candidates by looking up using the fixed value x, i.e., executing *lookup(x)*, it may happen that the querying node might already have candidates for the two variables. Therefore, the queried node can reduce the results sets with the knowledge of sets v_2 and v_3. However, those sets may be large, that is why authors use Bloom filters to reduce the representation of the sets. The counterpart of using Bloom filters, is that they yield false positives. Consequently, the final results sets which are transferred may contain non-matching results. To remove these candidates and thus ensuring the correctness of the query results, a final refinement iteration is done locally.

Liarou et al.

Liarou *et al.* propose in [95] two query processing algorithms to evaluate conjunctive queries over structured overlays, called Query Chain (QC) and Spread by Value (SBV).

- **Query Chain**
  - **Data indexation**

As in RDFPeers [89], a RDF triple is indexed three times. More precisely, for a peer p that wants to publish a triple t such as $t = (s, p, o)$, the index identifiers of t are computed by applying a hash function on s, p and o. Identifiers $hash(s)$, $hash(p)$ and $hash(o)$ are used to locate nodes n_1, n_2 and n_3 that will then store the triple t.

– **Query processing**

In this algorithm, the query is evaluated by a chain of nodes. Intermediate results flow through the nodes of this chain and the last node in the chain delivers the result back to the query's originator. More formally, the query initiator, denoted by n, issues a query q composed of q_1, q_2,..., q_i patterns and forms a query chain by sending each triple pattern to possibly different nodes, based on the hash value of constant part of each pattern. For each of the identified nodes, the message $QEval(q, i, R, IP(x))$ will be sent such that q is the query to be evaluated, i the index of the pattern that is managed by the target node, R a collection to hold intermediate results and IP the address of the query's originator x (i.e. the node that submits the query). When there is more than one constant part in the triple pattern, subject will be chosen over object and over predicate in order to determine the node responsible for resolving this triple. While the query evaluation order can greatly affect the algorithm performance including the network traffic and the query processing load, authors adopt the default order for which the triple patterns appear in the query.

- **Spread by Value**

  – **Data indexation**

  In the SBV algorithm, each triple $t = (s, p, o)$ is stored at the successor nodes of the identifiers $hash(s)$, $hash(p)$, $hash(o)$, $hash(s+p)$, $hash(s+o)$, $hash(p+o)$ and $hash(s+p+o)$ where the + operator denotes the concatenation of string values. By multiple indexation of the same triple, the algorithm aims to achieve a better query load distribution at the expense of more storage space.

– **Query processing**
SBV extends the QC algorithm in the sense that a query is processed by multiple chains of nodes. Nodes at the leaf level of these chains will send back results to the originator. More precisely, a node submitting a conjunctive query q in the form of $q_1 \wedge \ldots \wedge q_k$ sends q to a node n_1 that is able to evaluate the first triple pattern q_1. From this point on, the query plan produced by SBV is created dynamically by exploiting the values of the matching triples that nodes find at each step. As an example, a node n_1 will use the values found locally that matches q_1, to bind the variables of $q_2 \wedge \ldots \wedge q_k$ that are in common with q_1 and produce a new set of queries that will jointly determine the answer to the query's originator. Unlike the query chain algorithm, to achieve a better distribution of the query processing load, if there are multiple constants in the triple pattern, the concatenation of all constant parts is used to identify nodes that will process the query.

CumulusRDF

In [96] the authors investigate the applicability of a key-value store for managing large quantities of RDF data. Their solution dubbed Cumulus RDF is based on Apache Cassandra [35] which is a nested key-value store that belongs to the NoSQL movement which has emerged these last years. Cassandra's data model relies on column families, rows, columns and, optionally supercolumns. A column family depicts a table from the relational world. Inside a table we find rows that embed one or two level of nested key-value pairs depending of whether columns or supercolumns are employed. Thus, a simple row with columns looks like `{row_key: {column_key: column_value}}` whereas a row with supercolumns adds one extra level of key-value pairs as represented by `{row_key: {supercolumn_key: {column_key: column_value}}}`. Cassandra uses consistent hashing to distribute data. Consequently, the system exhibits the same topology as Chord but also features and maintenance algorithms that are similar to the Chord overlay network. Rows are assigned to nodes by hashing the row's key and storing the whole row, including associated columns and supercolumns, on the node which is

the closest in the identifier space. Another simple representation of the Cassandra model is to think about nested key-value pairs as a map of maps where the values of the first map are distributed across the nodes according to their key.

The solution proposed by the authors for CumulusRDF follows two strategies: one based on a hierarchical layout and another that relies on a flat layout. Both strategies make use of Cassandra properties which have been introduced above.

- **Hierarchical layout**
  - **Data indexation**
    The indexing scheme of the hierarchical layout is built on supercolumns. Each triple is indexed three times to provide the minimum indexes that are required to execute efficiently all eight possible triple patterns [97]. Each index is a Cassandra row with supercolumns where the row key, the supercolumn key and the column key are respectively RDF terms of the triple to index. To index the triple $t = (s, p, o)$, the rows `{s: {p: {o: -}}}` for the index SPO, `{p: {o: {s: -}}}` for the index POS and `{o: {s: {p: -}}}` for the index OSP are inserted in their dedicated column family according to the type of the index.
  - **Query processing**
    A triple pattern is evaluated by using one of the three indexes according to the fixed parts the triple pattern contains. For instance, to resolve the triple pattern $(?s, p, o)$, the index POS is used. By hashing the fixed value p, the node that stores the matching supercolumns is identified. Then, on that node a local lookup is performed with the fixed value p on the column key. As a result we get the columns' values matching the initial triple pattern. These values are object RDF terms of triples that match the fixed p value from the triple pattern. Thus, a local and final filtering based on o is performed in order to identify matching triples.
- **Flat layout**
  - **Data indexation**
    The second indexing strategy proposed by the authors is based on sim-

ple columns and the key observation that columns' key are sorted according to their natural order. Therefore, it is possible to perform range scans and prefix lookups on them. For this reason, they propose to store indexes as `{s: {po: -}}`, `{p: {os: -}}` and `{o: {sp: -}}`. However, they notice a complication with the POS index due to the fact that some properties such as *rdf:type* or *rdfs:label* are frequent in RDF. Rows are distributed across the nodes based on the row key, thus when the row key is a predicate value, a few nodes may have to store most of the triples from the whole system once. To alleviate this issue they propose to replace the POS index with two others where the row key is the concatenation of the predicate and object RDF terms. Since less triples share predicate and object, the distribution is better. These two indexes called POS_1 and POS_2 are respectively `{po: {s: -}}` and `{po: {'p': p}}`. The last index POS_2 maps column values to row keys so that it is possible to retrieve all PO row keys for a given p, `'p'` being an hardcoded string value.

– **Query processing**
Similarly to the hierarchical layout, the index to use is identified according to the fixed parts of the triple pattern to evaluate. To resolve $(s, ?p, ?o)$, the index SPO is used. The node containing solutions is found by hashing the subject RDF term and the columns found are directly returned. However, the solution for triple patterns that involve POS indexes is a bit more complex to compute. For instance, to resolve $(?s, p, ?o)$ the index POS_2 is required since rows may be found with hash based lookups only and this require to know the full row key value to hash it. Thus, a broadcast to all nodes is performed to find nodes indexing rows that contain potential solutions. Then, the POS_2 index used to filter rows that match the fixed value p.

Both strategies have been evaluated on 4 nodes in a virtualised infrastructure. The authors show their flat layout outperforms the hierarchical one with a factor of almost two in terms of concurrent requests handled per second. This at the price of more storage space since the hierarchical layout requires three indexes and

the flat layout one more. Besides, the choices made in CumulusRDF are driven by the need to retrieve RDF data by triple patterns and no discussion is given about the execution of complex SPARQL queries. Also, the layouts proposed in CumulusRDF may not work depending of the dataset that is considered. Indeed, the authors propose to create indexes and to store RDF term values in column or supercolumn keys. However, in Cassandra keys must be under 64 KB[2]. Even if the size is acceptable for subject or predicate values whose their IRI representation is not so large, the solution is inadequate for object values whose size may exceed this limit, especially if a subject or predicate value is concatenated with an object term.

Summary

A fair number of solutions have been proposed these last years to manage RDF data. With the great success the Semantic Web movement has achieved and the increasing amount of data to process, solutions have naturally evolved from a centralized environment to a distributed one. To address issues that are inherent to distributed systems, many works rely on P2P technologies and especially on structured P2P networks. However, with the advent of the NoSQL movement, a broad set of efficient systems are now available to manage information in the context of Big Data. NoSQL systems are an alternative to relational databases that usually sacrifice query complexity and ACID properties for more predictable query performance and low latency read/write operations. Since 2009, NoSQL systems are becoming more and more mature and the large variety of solutions has led to a classification in mainly four categories that appeared progressively: key/value oriented stores (e.g. Cassandra [35]), columns oriented stores (e.g. HBase [98]), documents based stores (e.g. MongoDB [99]) and graph oriented stores (e.g. AllegroGraph [100]). Each category has its advantages and drawbacks. Although NoSQL stores belong to a new movement, many rely on well known technologies such as P2P networks to manage distributed resources, especially because they use consistent hashing to distribute data. This is for instance the case with Cassandra which features a structure similar to Chord. However, some other NoSQL stores

[2] `http://wiki.apache.org/cassandra/FAQ#max_key_size`

such as HBase make use of a master/slave architecture. HBase is built on top of HDFS [101]. In contrary to HDFS which is a Distributed File System (DFS) designed for the storage of large files, HBase provides fast record lookups and updates. To make it possible, HBase internally puts data in indexed "StoreFiles" that exist on HDFS for high-speed lookups.

A main difference between Cassandra and HBase is that the former is fully decentralized while in the latter a master node, that is a single point of failure, is used. Also, Cassandra offers tunable consistency whereas HBase provides only strong consistency but with the benefit of a tight integration with the Hadoop ecosystem. Even though strong consistency means slower operations throughput than systems that provide eventually consistency, and thus is not appropriate for live queries, such systems remain acceptable to execute SPARQL queries over large datasets. Indeed, both solutions, regardless of their architecture are used daily in production with immensely large workloads, successfully.

These industry approved systems have been used recently in some research work to build RDF data managements systems. In DSPARQ [102], the authors propose to combine MapReduce with MongoDB [99]. Their solution spreads triples across the machines by using a graph partitioner. Although distributed query evaluation is supported, triples declaring *rdf:type* as predicate value are dropped, thus loosing data meaning and reducing reasoning capabilities. In H_2RDF [103] the authors leverage the HBase NoSQL store with its MapReduce interface to execute SPARQL queries over large RDF datasets. However, they are currently unable to support all features from the SPARQL specification. The strength of their system lies in the fact they provide an adaptive choice among centralized and distributed join execution for fast query responses. Recently, in [104] the authors give a fair comparison of four existing NoSQL solutions for processing RDF data. They consider results obtained with the systems they have analyzed encouraging since they are competitive against distributed and native RDF stores with respect to query time when simple SPARQL queries with no complex filters are used. Moreover, the authors are confident about the future of NoSQL databases as an alternative to native RDF engines to store and manage RDF resources because they think there is still many query optimization techniques that could be borrowed from relational databases and applied. Finally, some graph oriented solutions such

as AllegroGraph [100] or Bigdata [105] have been proposed but most of them are commercial and a few details about their architecture is available.

3.2 P2P Infrastructure for RDF

The brief state of the art we have presented gives an idea about the solutions and the different systems or topologies that have been proposed or extended to support RDF data. This section introduces the solution we have implemented, based on the approach proposed by Imen Filali [73], a former team member. Our solution differs from existing ones by its indexing process that does not rely on hashing. It explains, in part, why we do not have based our solution on one of the aforementioned solutions that intensively rely on hashing. Additionally, MapReduce operations introduce a high latency in data analysis which is not acceptable for extending our system to a publish/subscribe one as we will explain in Chapter 4. Also, contrary to existing solutions that usually decompose a SPARQL query into subqueries and resolve them through an execution plan that executes each subquery sequentially, we propose to handle subqueries in parallel with the aim to increase the throughput, adding when needed a final synchronization merging operation on results. For this purpose, we decided to rely upon the CAN network to deal with RDF information. In the following, we describe the CAN protocol before explaining what are the features and properties we altered. Then, we explain how generic messages are routed in our revised CAN infrastructure. Finally, we focus on the indexing and retrieval of RDF data through SPARQL and detail the solutions we propose along with their benefits and drawbacks.

3.2.1 Content Addressable Network (CAN)

CAN [20] is a Structured Overlay Network where the peers that compose the network are virtually organized on a d-dimensional Cartesian coordinate space labeled $\mathcal{D}$. The coordinate space is dynamically partitioned among all peers in the system such that each node is responsible for storing data in a zone of $\mathcal{D}$.

Each zone is an hyperrectangle defined by exactly one upper bound and one lower bound coordinate value in $\mathcal{D}$. Since the coordinate space is entirely parti-

tioned, the zones abut each others in one or more dimensions. The set of peers that abuts a given peer p is called the neighbourhood of p. Figure 3.2 depicts a two-dimensional CAN network made of 4 peers arranged in a $[0,1] \times [0,1]$ coordinate space. Geometrically speaking, two peers are neighbours if their edges overlap in exactly $d-1$ dimensions and abut in exactly 1 dimension.

CAN provides, like many structured P2P networks, a DHT abstraction. Thus, resources are indexed by keys. Keys are computed by applying for instance d different uniform hash functions on the resource value to index. In this manner, each resource is associated to a key that is a coordinate $C = (h_1(value), \ldots, h_d(value))$ which depicts a point in $\mathcal{D}$. For a uniformly partitioned space with n nodes and d dimensions, a CAN network exhibits the following properties. Each peer maintains $2d$ neighbours and the average routing path length to route a message is $(d/4) \times (n^{1/d})$. A CAN network can also achieve the same scaling properties as other popular DHTs such as Chord [21] and Pastry [22], by routing in $O(\log n)$ hops if the number of dimensions is set to $d = (\log_2 n)/2$.

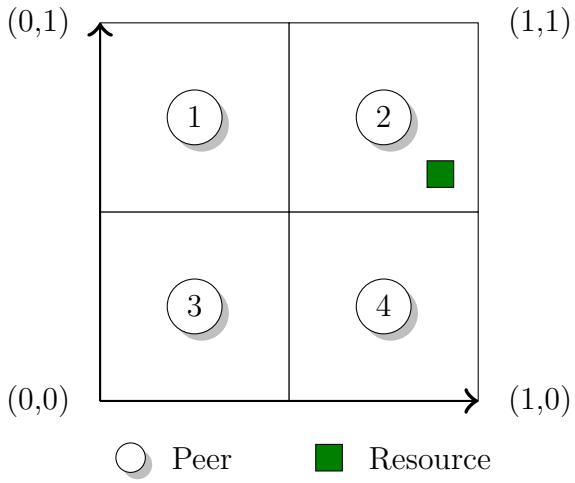


Figure 3.2 – Simple 2-dimensional CAN network.

Message routing The routing algorithm that is proposed by CAN's authors assumes that messages and more specifically requests are routed according to keys. A key is, as introduced previously, a coordinate in the d-dimensional space. The routing algorithm the authors have designed allows to send an arbitrary request

to the unique peer that manages the key used for the routing. For example, if the purpose is to route a message to store a resource, only the peer that manages the resource key will perform the final operation that consists of storing the resource. We summarize hereafter the process that starts with a coordinate C for a key k associated to a resource with a resource value v. First, a peer p_i is selected at random to receive the message M to route. Once p_i receives M, it applies a simple greedy forwarding approach. The algorithm consists of choosing the closest peer to C from the neighbours set managed by p_i. The selection is done after the computation of the euclidean distance from the center of each neighbour's zone to the coordinate C. The peer p_j that is selected as the one with the smallest distance value receives the message forwarded by p_i. Then, the process is repeated until to reach a peer p_{final} whose its zone contains C. At this step, a response that embeds for example v may be returned by using the same routing algorithm if the purpose was to find a resource by key. Otherwise if the goal was to index a resource, the key-value pair (k, v) is simply stored locally by p_{final}.

Join procedure The join procedure allows a new peer to join an existing one. Let say that p_{new} is a peer to insert into the network. Then, the join procedure works as follow. First, a coordinate C is picked at random from the coordinate space $\mathcal{D}$. Afterwards, a join request is forwarded towards the peer managing C by using the routing algorithm described above. The peer reached with the join request, denoted by $p_{landmark}$, is the peer that has to divide up its zone with p_{new}. Thus, $p_{landmark}$ splits its zone in two and gives one half to p_{new} along with the resources managed by the zone chunk. Finally, the neighbourhood of $p_{landmark}$ and p_{new} is updated.

Leave procedure The leave procedure allows a peer to leave gracefully a CAN network it has previously joined. For that, the zone z_{leave} managed by the peer p_{leave} that leaves must be taken over by one of the remaining peers from the network. In this respect, the authors distinguish two cases. Either z_{leave} may be merged with a zone z_{valid} that belongs to a neighbour p_{valid} of p_{leave} in such a way that the resulting zone is still an hyperrectangle, or it can not. In the former case, the merging is done and the resources are forwarded from p_{leave} to p_{valid}.

However, in the latter case, z_{leave} is handed over to the peer with the smallest zone volume from p_{leave}'s neighbourhood. The selected peer will temporarily manage two independent zones. Then, a background zone reassignment process, that may imply to merge several zones in chain and thus to transfer multiple key-value pairs, is assumed to reassign zones and ensure that the CAN network tends back towards one zone per peer. Each peer zone remaining an hyperrectangle.

Our revised CAN settings

Most of the related works presented formerly use hash functions to index RDF data. We have seen this is also the case by default with CAN for indexing resources. However, hashing incurs a storage overhead when RDF tuples are manipulated since they are stored multiple times to be found back in an efficient manner with triple or quadruple patterns. In addition, one big drawback that is intrinsically linked to hash functions is that it is really difficult to support range queries (looking for values in a specified range) efficiently. There is an exception with locality preserving hash functions, also referred to as Locality Sensitive Hashing (LSH), but since similar items remain close together once hashed, systems using it are facing load imbalance issues that are most of the time underestimated or ignored. This along with an extra level of complexity incurred by hashing. To address these disadvantages, we propose a new distributed RDF datastore architecture that mimics the natural format of RDF tuples and which does not rely on hash functions.

Our infrastructure is built atop the CAN overlay to ensure its scalability. More precisely the architecture rely upon a four-dimensional CAN network. The four dimensions of the CAN coordinate space are mapped respectively to the graph, the subject, the predicate and the object RDF terms of any RDF 4-tuple that may be indexed. One benefit of this natural approach, that reflects the structure of an RDF quadruple, is that a quadruple to index represents a coordinate and thus a point in the four-dimensional Cartesian space. Moreover, as we will further explain, the indexing does not make use of hash functions. Quadruples are routed to the peers that manage the quadruple coordinates by means of the lexicographic order applied on the quadruples' RDF terms. Therefore, the CAN network is no

longer a DHT but a distributed lexicographically ordered data structure. This approach has several advantages. First, it enables to process range queries efficiently. Second, the lexicographic order preserves the data semantics so that it gives a form of clustering of quadruples sharing a common prefix, thus improving lookup. In contrast, hashing destroys the natural ordering of information and makes the management of complex queries tricky and expensive.

To support the lexicographic order, in our revised CAN overlay, the bounds of the CAN network are unicode characters. Unicode characters are numbers encoded in 32 bits integers, also referred as codepoints, whose their integer representation may range from 0 to $10FFFF_{16}$ = 1114111. For instance, the characters *'A'* and *'Z'* have respectively the codepoint values 65 and 95. The fact that coordinate values are strings made of Unicode characters affects some operations related to the CAN protocol since now coordinates are made of numbers in radix different from 10. Operations such as the split operation performed during a join or even the operation that computes the euclidean distance between two coordinates require an update to work with any radix n. Also, to mitigate the impact of skewed data when specific information is known about the data distribution, we allow to define at startup the lower and upper bounds of the CAN space. For example, the bounds may be defined to allow only the CJK Unicode block that corresponds to Chinese ideographs usually used in the writing systems of the Chinese and Japanese languages, occasionally for Korean, and historically in Vietnam. In case we know that all RDF terms are expressed in this CJK block, an extra shift condition has to be taken into account in operations working in radix n.

3.2.2 Routing algorithms

In P2P systems, information is spread among peers and expensive computations aim to be shared between peers. The indispensable substrates to distribute data, and work in general, are messages which are routed to peers according to a key before being executed locally. Message routing strongly depends on messages' type and the kind of key associated to them. Also, message routing directly affects the overall performance of operations. For this reason we detail hereinafter the routing algorithms we are using in our revised CAN overlay.

The P2P infrastructure we have designed is aiming to support RDF data. In such a system the main operations are the insertion and the lookup of data. It is worth notice here that these two operations require messages with different routing processing. The reason lies in the fact that for the insertion, the message has to reach only one peer, the one that manages the key or coordinate that contains RDF terms from a quadruple to index. However, when data have to be retrieved, the message has to get to one or more peers since the different pieces of information we are interested in may potentially be located on multiple different peers. For that purpose, we have designed two routing algorithms whose details are given below.

Unicast routing

Unicast routing aims to route messages whose associated key is a coordinate with fully fixed coordinate values. This type of key is referred in the rest of this dissertation as a unicast key. A simple approach to route these messages may be to reuse the default message routing algorithm proposed by the CAN's authors. However, as we explained previously, our CAN overlay make uses of Unicode characters as coordinate values and doing so requires to compute several times the euclidean distance between coordinates at each routing step with a radix different from 10 (i.e. radix 1114112). This is not conceivable since the routing will incur expensive local decisions. The reason lies in the fact that the euclidean distance involves mathematical operators such as the multiplication or the square root that are costly to compute with large radix. To prevent expensive computation, we introduce an algorithm that solves the issue by simply comparing zones' coordinates with the key used to route a message.

The simple unicast routing mechanism is sketched by Algorithm 3.1. The algorithm makes the assumption to route a message m from a peer p to the peer managing the key k which is a coordinate made of d coordinate values. The scheme is the following. Each dimension is iterated successively by the peer p that is currently routing the message. The peer starts from the first dimension. If the coordinate value $k[1]$ (which is the first coordinate value associated to k) is contained by p's zone on dimension 1, then the same checking is applied on

```
1: procedure UnicastRouting(m, p, k)
2:     for i ← 1, d do
3:         if !p.getZone().contains(k[i], i) then
4:             n ← p.findClosestNeighbour(k, i)
5:             UnicastRouting(m, n, k)
6:             return
7:         end if
8:     end for
9:     m.execute()
10: end procedure
```

Algorithm 3.1 – Unicast routing algorithm.

the next dimension until to have k fully managed by the peer's zone on all the d dimensions. In that case, the message action is executed and a response may be returned, either directly to the requester if its reference is known or by using the same unicast routing algorithm. However, if p's zone does not contain $k[i]$ on the i-th dimension, the message m is forwarded to p's closest neighbour that applies in its turn the same process. The function used to find the closest neighbour does not rely upon the euclidean distance. It simply finds in p's neighbourhood, the closest neighbour's zone by comparing the bounds managed by the current peer with coordinate values from k. At this stage, two cases may occur when the closest neighbour function is executed. Either there is only one neighbour n that manages $k[i]$ on dimension i or there are two or more neighbours. In the last case the conflict is resolved by selecting the neighbour with lexicographically the closest zone's bound to $k[i+1]$ on dimension $i+1$. Then, p forwards the message along with the key k to the selected neighbour n. Afterwards, this last applies again the unicast routing procedure until to have all the key's values managed by its zone.

The routing algorithm has been simulated and the results remain acceptable since the complexity in terms of number of hops follows the same trend as the one given by CAN's authors.

Multicast routing

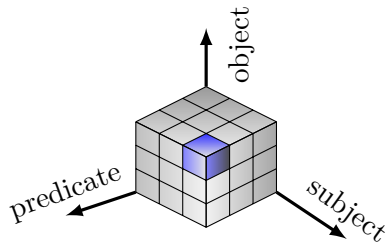
Unlike unicast routing whose purpose is to reach a unique peer, multicast routing allows to send a message to a set of peers. The destination is appointed with the

help of a key we refer to in this context as a multicast key. The specificity of multicast keys is that they allow free variables in their definition. Let's assume a simple example with a key $k = (?v_1, p_1, ?v_2, p_2)$ where $?v_1$ and $?v_2$ represent free variables and the other coordinate elements fixed values. In that case, k points out a set of peers G that manage respectively p_1 and p_2 on the first and third dimension. On other dimensions, these peers may manage any value since free variables are specified. Figure 3.3 depicts different set of peers to reach according to distinct definition of keys with free variables on a 3D CAN.

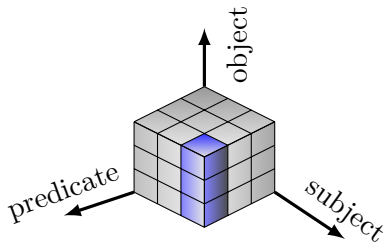
Another interesting point to notice here is that in our revised CAN overlay, we are using the lexicographic order on the coordinate values to route them to the right peers. Therefore, the set of peers to reach may be further restrained by extending free variables with the notion of bounded variables. This, with the goal to designate more precisely a set of peer we are interested in. For instance, the multicast key $(?v_1, p_1, ?v_2, p_2), ?v_1 \in [l, u]$ refers to a subset of G that possibly reduces the number of peers that satisfy the key's constraint by means of a bounded variable $?v_1$ that force peers' zone to contain a chunk of the interval $[l, u]$ on the first dimension to handle the message. Here, we assume that at most one variable is bounded. Cases with two, three or four bounded variables have to be decomposed into simple ones with one bounded variable each, thus requiring to route multiple messages and aggregate results at the application level. However, the increased cost to route a decomposed set of keys should be balanced by the fact that messages are routed and handled in our system in parallel.

Algorithm 3.2 details how multicast messages are routed based on a multicast key containing free variables only. The message starts from an initiator p which is selected at random. The first step consists of finding a peer whose its zone fully manages[3] the multicast key k (line 2–7). The statements are really similar to the unicast routing procedure introduced in Algorithm 3.1. Indeed, only the condition on line 3 differs. This line contains an additional predicate to consider any coordinate value, which is a variable, as managed by a peer's zone on any dimension. Once a peer n is detected as managing k, all other peers that satisfy k may be reached from neighbours to neighbours. This is possible thanks to

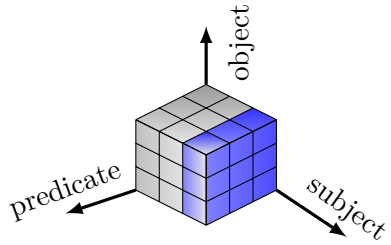
[3]In that case, a peer p is said to manage a key k if its zone contains all the fixed part of k and satisfies conditions associated to k.



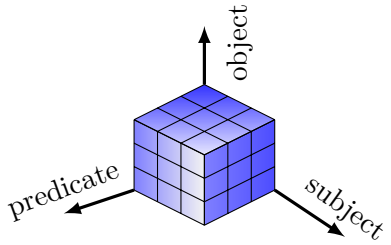
(a) Key values fixed on all dimensions.



(b) One free variable on the last dimension.



(c) Two free variables on the last two dimensions.



(d) Free variables on all dimensions.

Figure 3.3 – Multicast keys' scope according to various fixed terms in a 3-dimensional CAN network. The graph dimension has been intentionally omitted for the sake of the representation.

the definition of the multicast key's properties that allow multiple free variables but only one bounded variable. Consequently, p invokes the multicast procedure (line 11–18) that executes a custom action `onPeerValidatingKeyConstraints` to query for example a local datastore. Then, if some neighbours satisfy k's values and conditions, the multicast message is forwarded to them.

Responses are optional like for the unicast routing algorithm. To support them, a reverse path forwarding scheme must be applied. Each peer n which receives a request and is a leaf of the multicast tree (i.e. it does not forward the request to another neighbour) has to send back the response to the peer p from where the request was sent. Then, p merges the responses into one and sends it back to its predecessor. This is performed recursively until to have a final response that reaches the initiator. This approach is required and is due to the fact that at each response forwarding step, the peer p that receive responses is the only one to know how many requests it has sent and, therefore, how many responses it has to receive and merge before forwarding back an answer. Although we do not focus on fault tolerance, as a rule of thumb we never use a direct peer reference to send back a response. Such a method would require a major redesign to support fault tolerance. Instead, peers are identified by their lower left zone's bounds. Therefore, responses are routed back based on these coordinates that have been memorized.

It is also important to note that whenever a key with variables is processed, our approach naively uses message flooding through each peer's neighbours. Hence, it may happen that a peer receives a message multiple times from different dimensions. Even if in practice these duplicate messages are ignored, they, however, overload peers. Francesco Bongiovanni, a former team member, has worked on an optimal broadcast algorithm for CAN. He has proved there exists an algorithm that covers the whole CAN network without sending twice a message to the same node [106]. His solution is a generalization of the efficient broadcast algorithm proposed in M-CAN [107]. The general idea of the algorithm may be roughly summarized as follows: a peer forwards a message to a neighbour n if n touches the lower left zone's bound of the peer and if n satisfies a spatial constraint defined at the beginning of the algorithm. This optimal broadcast algorithm (in terms of message complexity) has been implemented and evaluated in our P2P infrastruc-

ture [108]. The experiments show that this scheme may prevent in average 750 duplicates to transit in a CAN network made of 100 peers. Also, this optimal approach reduces message delivery time compared to our naive one which generates duplicates. In average, the message delivery time is decreased by 6.25% with 100 peers and up to 41.5% with a CAN overlay made of 500 peers.

```
1: procedure MulticastRouting(m, p, k)
2:     for i ← 1, d do
3:         if !k[i].isVar() ∧ !p.getZone().contains(k[i], i) then
4:             n ← p.findClosestNeighbour(k, i)
5:             MulticastRouting(m, n, k)
6:             return
7:         end if
8:     end for
9:     Multicast(m, p, k)
10: end procedure
11: procedure Multicast(m, p, k)
12:     if m not already received then
13:         m.onPeerValidatingKeyConstraints()
14:         for each n ∈ p's neighbourhood do
15:             if n.getZone().satifies(k) then
16:                 Multicast(m, n, k)
17:             end if
18:         end for each
19:     end if
20: end procedure
```

Algorithm 3.2 – Multicast routing algorithm.

However, the optimal broadcast algorithm remains limited in our context since multicast requires to flood the whole network, even if the action is only executed on peers validating the multicast key constraints. Indeed, with the optimal broadcast scheme, multiple unique propagation paths are created and to cut off some according to multicast constraints may prevent one or more peers to receive the message they should handle.

3.2.3 Indexing and retrieval mechanisms

In this section we illustrate how the routing algorithms we presented previously are used in action to index RDF quadruples but also to execute SPARQL queries.

RDF data indexing

To index RDF data (i.e. to route and store a quadruple to the right peer) we rely upon the unicast routing algorithm. We define a unicast key based on the quadruple's values and route an index quadruple message. For the sake of the explanation, consider the Figure 3.4. In this representation each peer manages a zone whose the bounds are denoted per dimension by $z_{g_{min}}$ and $z_{g_{max}}$ for the graph value of a quadruple, $z_{s_{min}}$ and $z_{s_{max}}$ for the subject value, $z_{p_{min}}$ and $z_{p_{max}}$ for the predicate value, and finally $z_{o_{min}}$ and $z_{o_{max}}$ for the object value. In our terminology we say that a 4-tuple $q = (g, s, p, o) \in z$ if and only if $\forall$ RDF Term r $\in$ q, $z_{r_{min}} \leq r \prec z_{r_{max}}$.

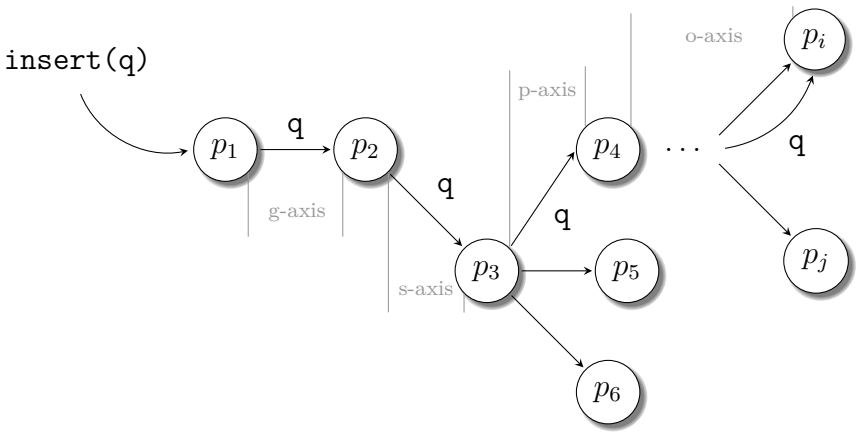


Figure 3.4 – Exemplary series of actions to insert a quadruple into a 4-dimensional CAN.

Now, suppose that a peer p_1 receives an `insert(q)` request aiming to insert the RDF quadruple to the network. To find the peer where the quadruple will fall in, a greedy scheme is applied. Locally, the peer p_1 checks whether an element of q is contained by its zone or not. The comparison starts with the first RDF term of q, that is to say the graph value. Since no element of q belongs to the zone of p_1, and as g fits into the zone of p_2, p_1 routes the insert message to its neighbour

p_2 according to the graph axis (g-axis). The same process is triggered at each stage. The peer p_2 does it by means of the subject axis (s-axis) which in turn, will forward the message to its neighbour p_3. Once received, p_3 checks whether one of its neighbours is responsible for a zone such as q belongs to, and sends q to p_4 through the p-axis. Finally p_4 examines its zone for o. Since no immediate neighbor is managing the o value, the message is forwarded according to the object axis (o-axis) to the neighbour with object coordinates that are closest to o, ending up on peer p_i that is responsible for storing q locally.

SPARQL query execution

Insertion but also retrieval are the more common operations that are usually performed on a P2P network. Many systems such as Chord [21], Pastry [22] or even CAN [107] rely by default on consistent hashing to uniformly spread out keys onto peers at the cost of complex and expensive mechanisms to support range queries. In contrary to these DHTs, our revised CAN overlay natively supports range queries since we rely on the lexicographic order to route messages. However, our system, as the others, does not handle by default complex requests such as SPARQL queries. Therefore, to execute a SPARQL query q, part of the process consists in analysing q to extract subqueries. Subqueries are elementary requests that are handled efficiently by routing specific messages in our system. Similarly to RDFPeers [89], we have identified atomic and range queries as subqueries. An atomic query is a quadruple pattern from a SPARQL query that maps to a multicast key with one or more free variables according to the quadruple pattern's variables. A range query extends the atomic query notion by allowing at most one bounded variables. Intuitively, this second type of query maps also to a multicast key as introduced in Section 3.2.2. More complex query types, such as conjunctive or disjunctive queries are supposed to be decomposed into atomic or range ones.

The full mechanism to execute a SPARQL query is sketched by Algorithm 3.3. The execution starts from a peer p selected at random. Once the SPARQL query q has been decomposed into subqueries (line 2), each subquery is handled in parallel to others. For instance, if we are handling a subquery s, first a message m with a multicast key k is created according to s' values and conditions (line 4–

5). Then, m is routed from p (line 6) to the peer(s) that manage the multicast key k based on the multicast algorithm introduced previously. Upon the reception of m on a neighbour n that validates k's constraints, a background thread is spawned to query n's local RDF store by means of a SPARQL query dynamically built from k. This background action is made possible by overriding the method `onPeerValidatingKeyConstraints` from Algorithm 3.2. Since tuples are retrieved in a background thread, the routing algorithm may continue to forward m to other neighbours without waiting for the retrieval of tuples matching k. However, results have to be recovered. Fortunately, as we explained in Section 3.2.2, the reverse forwarding path is used to route a response. Consequently, a response reaches the peers where a local query has been triggered in background. At this step, the termination of the background thread is awaited. The aim is to attach to the response the tuples that have been found with the local query before it is routed back. Finally, once all results from subqueries have been collected into a tuples set R, a final filtering operation is applied with the help of a *Strain* function. Its purpose is to filter R with the original SPARQL query q in order to resolve SPARQL operators which have not been handled in a distributed manner (e.g. join conditions between subqueries due to BGPs or regular expressions).

```
1: function ExecuteSparqlQuery(q)
2:     subqueries ← Decompose(q)
3:     for each s ∈ subqueries do in parallel
4:         m ← CreateAtomicMessage()
5:         k ← CreateKey(s)
6:         result ← Route(m,p,k)
7:         R ← R ∪ {result}
8:     end for
9:     return Strain(R, q)
10: end function
```

Algorithm 3.3 – SPARQL query execution algorithm.

To better illustrate what we explained, let assume a concrete example with bibliographic resources described in RDF (as in Section 2.2.1) and stored in our system. A possible query could be to find all authors' firstname who have published bibliographic resources in 1987 or between 2010 and 2013, so that their firstname

starts with the letter E and ends with the letter r. Listing 3.1 represents this SPARQL query.

The decomposition for the aforementioned SPARQL query results in 5 subqueries. On one side there are four atomic queries which are respectively $(?g, ?isbn, rdf : type, dc : BibliographicResource)$, $(?g, ?isbn, dc : creator, ?creator)$, $(?g, ?isbn, dc : date, 1987)$ and $(?g, ?creator, dc : firstName, ?firstName)$ since we do not leverage regular expressions at the routing level. On the other side there is one range query for $(?g, ?isbn, dc : date, ?date)$, 2010 ≤ ?date < 2014. Similarly to the *scan* phase of centralized RDF stores, tuples that satisfy each subquery are retrieved, in parallel. Afterwards, a final filtering is applied on the resulting tuples set with the initial SPARQL query. This last step may be seen like the usual *join* phase from centralized stores. However, in our case its purpose is more general since it allows us to transparently support operators that are not handled at the routing level such as the regular expression on line 7.

```
1  SELECT ?firstName WHERE {
2      GRAPH ?g {
3          ?isbn rdf:type dc:BibliographicResource .
4          ?isbn dc:date ?date .
5          ?isbn dc:creator ?creator .
6          ?creator dc:firstName ?firstName .
7          FILTER(REGEX(?firstName, "^E.*r$")
8              && (?date = 1987 || ?date >= 2010 && ?date < 2014))
9      }
10 }
```

Listing 3.1 – SPARQL query example for retrieving authors' firstname with conditions on the publication date of bibliographic resources and the letters contained in authors' firstname.

3.3 Evaluation

In order to validate our P2P infrastructure for RDF data, we have performed micro benchmarks on the French Grid'5000 testbed. The goal was twofold. First, we wanted to evaluate the overhead induced by the distribution and the various

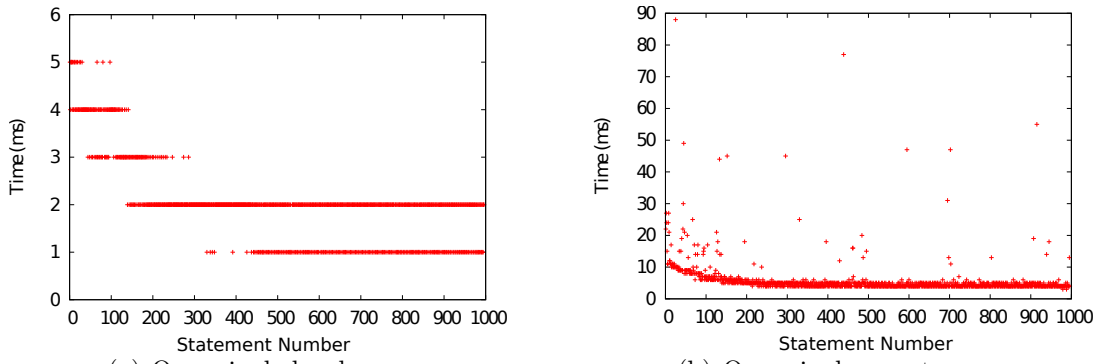
software layers between the repository and the end user. Second, we wanted to evaluate the benefits of our approach, namely the scalability in terms of concurrent access and the overlay size.

All the experiments presented in this section have been performed on a 75 nodes cluster with 1Gb Ethernet connectivity. Each node has 16GB of memory and two Intel L5420 processors for a total of 8 cores. For the 300 peers experiments, there were 4 peers and 4 RDF repositories per machine, each of them running in a separate Java Virtual Machine.

3.3.1 Insertion of random data

Single peer insertion

The first experiment performs 1000 statements insertion and we measured the individual time for each of them, on a CAN network made of a single peer. The two entities of this experiment, the caller and the peer, are located on the same host. The commit interval was set to 500 ms and 1000 random statements were added. Figure 3.5(a) shows the duration of each individual call. On average, adding a statement took 2.074 ms with slightly higher values for the first insertions, due to cold start.



(a) On a single local peer.

(b) On a single remote peer.

Figure 3.5 – Local vs remote insertion on a single peer.

In a second experiment, the caller and the peer were put on separate hosts in order to measure the impact of a local network link on the performance. As shown

in Figure 3.5(b), almost all add operations took less than 9 ms while less than 6.7% took more than 10 ms. The average duration for an add operation was 6 ms.

Multiple peers insertion

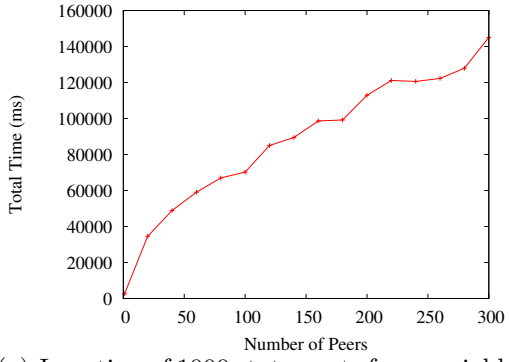
We have measured the time taken to insert 1000 random statements in an overlay with different number of peers, ranging from 1 to 300. Figures 3.6(a) and 3.6(b) show respectively the *overall* time when the calls are performed using a single or 50 threads. As expected, the more peers, the longer it takes to add statements since more peers are likely to be visited before finding the correct one. However, when performing the insertion concurrently, the total time is less dependent on the number of peers. Depending on the various zones sizes and the first peer randomly chosen for the insertion, the performance can vary, as can be seen with the small downward spike on Figure 3.6(b) at around 80 peers. To measure the benefits of concurrent access, we have measured the time to add 1000 statements on a 300 peers overlay while varying the number of threads from 1 to 50. Results in Figure 3.6(c) show a sharp drop of the total time, clearly highlighting the benefits of concurrent access.

3.3.2 Queries using BSBM

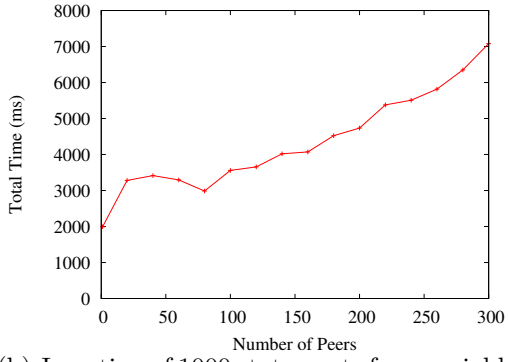
The Berlin SPARQL Benchmark (BSBM) [109] defines a suite of benchmarks for comparing the performance of storage systems across architectures. The benchmark is built around an e-commerce use case in which a set of products is offered by different vendors, with given reviews by consumers regarding the various products. The following experiment uses BSBM data with custom queries detailed below. The dataset is generated using the BSBM data generator for 666 products. It provides 250030 triples which are organized following several categories: 2860 Product Features, 14 Producers and 666 Products, 8 Vendors and 13320 Offers, 1 Rating Site with 339 Persons and 6660 Reviews.

Custom queries executed

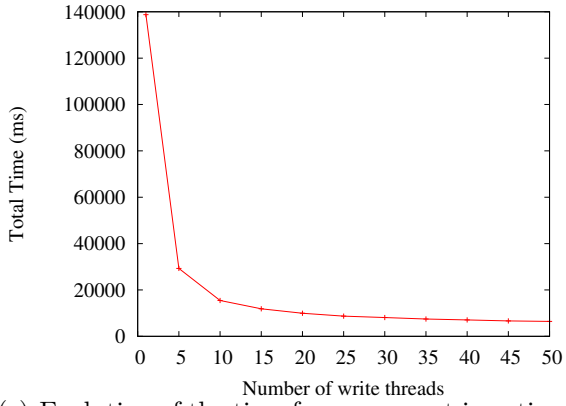
Owing to the fact that we support efficiently only a subset of SPARQL, we chose out of this benchmark specification, four queries which are executed independently



(a) Insertion of 1000 statements for a variable number of peers with 1 thread.



(b) Insertion of 1000 statements for a variable number of peers with 50 threads.



(c) Evolution of the time for concurrent insertions with 300 peers.

Figure 3.6 – Sequential and concurrent insertions with up to 300 peers.

by using a different initial peer each time. These queries use the namespaces referenced in Table 3.1.

Q1 – Returns a graph where producers are from Germany:

```
CONSTRUCT {
    iso:DE <http://www.ecommerce.com/Producers> ?producer
} WHERE {
    ?producer rdf:type bsbm:Producer.
    ?producer bsbm:country iso:DE
}
```

Q2 – Returns a graph with triples containing instances of *purl:Review*:

```
CONSTRUCT {
    ?review rdf:type purl:Review
} WHERE {
    ?review rdf:type purl:Review
}
```

Q3 – Returns a graph where triples imply a *rdf:type* relation as predicate:

```
CONSTRUCT {
    ?s rdf:type ?o
} WHERE {
    ?s rdf:type ?o
}
```

Q4 – Returns a graph where *bsbm-ins:ProductType1* instance appears:

```
CONSTRUCT {
    bsbm-ins:ProductType1 ?a ?b
    ?c ?d bsbm-ins:ProductType1
} WHERE {
    bsbm-ins:ProductType1 ?a ?b .
    ?c ?d bsbm-ins:ProductType1
}
```

bsbm	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
bsbm-ins	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
iso	http://downlode.org/rdf/iso-3166/countries#
purl	http://purl.org/stuff/rev#

Table 3.1 – BSBM namespaces used by the queries considered.

Q1 and Q4 are complex queries and will be decomposed into two subqueries. Hence, we expect a longer processing time for them. The number of matching tuples is given in Table 3.2.

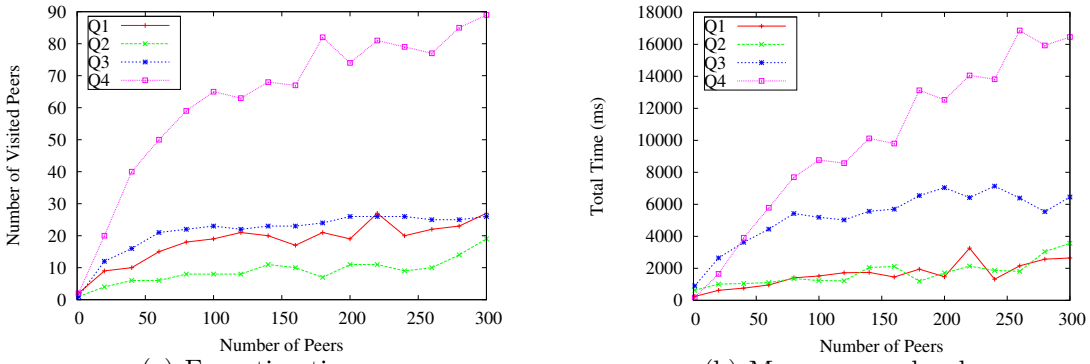
Query	Q1	Q2	Q3	Q4
# of results	1	6660	25920	677

Table 3.2 – Number of final results for the queries considered.

Figure 3.7 shows the execution time and the number of visited peers for processing Q1, Q2, Q3 and Q4. Note that when a query reaches an already visited peer, it will not be further forwarded, therefore we do not count it. Q1 is divided into two subqueries with only a variable subject. Hence, it can be efficiently routed and is forwarded to a small number of peers. Q2 also has one variable and thus exhibits similar performance. Q3 has two variables so it will be routed along two dimensions on the CAN overlay, reaching a high number of peers. Since it returns 25920 statements, the messages will carry a bigger payload than for the other queries. Finally, Q4 generates two subqueries with two variables each, making it the request with the highest number of visited peers. On the 300 peers network, the two subqueries have visited more than 85 peers.

Concluding remark Regarding tuples insertion into the distributed storage, although a single insertion has a low performance, it is possible to perform them concurrently, leading to a higher throughput. The performance of queries is more complex to predict since it depends on the number of subqueries, the payload carried between peers and the number of visited peers. While the payload depends on the request itself (number of variables and constraints), the number of peers depends not only on the structure of the overlay but also on the randomly chosen

peer for initiating the query.



(a) Execution time. (b) Messages overhead.

Figure 3.7 – Custom queries with BSBM dataset on various overlays.

Summary

In this chapter we have presented a distributed RDF storage based on a structured P2P infrastructure. RDF tuples are mapped on a four-dimensional CAN overlay based on the value of its elements. The global space is partitioned into zones and each peer is responsible for all the tuples falling into it. We do not use hash functions, thus preserving the data locality. SPARQL queries are decomposed into subqueries that are executed in parallel. We have validated our implementation with micro benchmarks. Although basic operations like adding tuples suffer from an overhead, the distributed nature of the infrastructure allows concurrent accesses. In essence, we trade performance for throughput.

Obviously, our solution has some drawbacks. The first downside is that our approach is sensitive to the distribution of data. Since we use the lexicographic order to index data, when some RDF tuples share the same namespaces or prefixes the probability they end up on a same peer is very high. Therefore, one or more peers might become a hot zone. To address this issue, we will introduce and discuss some solutions in Chapter 5. The second inconvenient with our solution is related to the execution of SPARQL queries. We decided to handle subqueries in parallel. However, when subqueries share common variables (i.e. require a

join) and return tuples set with sizes that differ from one ore more orders of magnitude, our solution requires to carry from peers to peers and until the query initiator many tuples that would be unnecessary if subqueries were pipelined by previously building a sequential query plan. This issue, that relates to the number of intermediate results to convey in the network in order to solve a SPARQL query has been highlighted empirically with our system [110]. Quilitz *et al.* propose in [111] to build a query plan that executes subqueries sequentially after they have been sorted in descending order according to the number of fixed parts and their position (i.e., graph, subject, predicate or object of a quadruple pattern). Indeed, subqueries with a lot of fixed part are assumed to return a few number of tuples. Consequently, the next subquery to execute may leverage the result from the previous one to reduce the number of intermediate results. One step forward, in [112] the authors propose to investigate the selectivity of subqueries (i.e. an estimation or an exact value about the number of tuples subqueries are expected to return once executed). This way an optimal query plan may be executed. In this respect, a perspective could be to combine our solution with an optimal query plan to execute queries. That, with the idea to still execute in parallel subqueries with a low selectivity or whose a bright reordering might not reduce the bandwidth consumption. Also, another point that hurts performance and that could be considered to enhance the execution time is related to the latency to route messages. Peers that are close together in the underlying network topology could be moved as neighbors at the overlay layer. Ali *et al.* have shown recently in [113] that a structured P2P system improved with locality awareness and some additional shortcuts for frequently used routes may boost performance by a factor of two in an RDF context.

Chapter 4

Distributed RDF Publish/Subscribe

Contents

This chapter details our second contribution that relates to a publish/subscribe layer for storing and selectively disseminating RDF events. It is built as an extension atop the infrastructure introduced in the previous chapter and relies on the routing algorithms that were described earlier. We start to compare existing solutions and we explain why building RDF-based event systems differs from conventional publish/subscribe systems. Then, we introduce our publish/subscribe

infrastructure for RDF events. First, we detail the event and subscription model suitable for RDF data we propose. Afterwards, we list the different properties or requirements our publish/subscribe system is assumed to respect, before entering into the details of two event filtering algorithms. Their characteristics and differences are explained, discussed and analyzed. To conclude, the algorithms we propose are evaluated in a distributed environment.

4.1 Related Works

Event-based systems are at the heart of publish/subscribe interactions. The ancestors of these systems are most probably Database Management System (DBMS) that were the first to instigate the need for reacting to data changes. In the rest of this section we briefly discuss solutions built on top of relational databases, also known as active databases, whereupon we enter into the details of some well known publish/subscribe systems and explain how they differ from RDF-based event systems. In both cases, the major difficulty in the presented solutions is to decide how events and subscriptions are indexed so that a matching is possible and notifications delivered, ideally without duplicates. Also, system responsiveness is a real challenge, especially depending on the expressivity of subscriptions. In this context, we will see what are the tradeoffs made by system's authors and how our solution is placed regarding to others.

4.1.1 Active databases

Solutions such as HiPAC [114], Ode [115], Postgres [116], that are known as Active Databases [117] rely on database triggers that are actions which are executed when an event occurs, such as the modification of a database row or table. The benefits of triggers are multiple. They are usually used to audit changes, replicate data or even enhance performances by summarizing values for future queries. However, as we already mentioned, DBMS has been designed some years ago when deployment was most of the time targeting a single machine. Also, triggers are applied at the schema level (i.e. when a row or table is created, edited or deleted) and do not achieve the exact same purpose as traditional publish/subscribe systems.

In their essence, database triggers are more generic than subscriptions from publish/subscribe systems. The reason lies in the fact that database triggers rely on Event-Condition-Action (ECA) rules that decouple and abstract the condition on which an action is executed and the action itself. In contrary to subscriptions that filter and forward matching events to their respective subscribers, ECA rules allow to express conditions based on database state values and to execute arbitrary actions when they are satisfied. However, to express conditions based on some state raises one main question which is how to ensure data consistency across different machines. Many solutions exist and all relational databases implement one since they focus on ACID properties. Nonetheless, to enforce consistency in a distributed system has the direct and bad effect of increasing operations latency, which is most of the time not acceptable in publish/subscribe systems since they aim to filter information in near real-time.

4.1.2 Conventional publish/subscribe systems

In the last two decades, the flexibility, modularity and responsiveness of publish/subscribe led to the emergence of several solutions. Broadly speaking, these systems are classified into *topic-based* or *content-based* categories according to their expressivity. Tibco [118] and Pubsubhubbub [119] are representatives of this former category that provides limited filtering capabilities. Most prominent solutions regarding the latter category are certainly Siena [120] and Hermes [55]. Siena, uses *covering-based* routing algorithms to reduce routing entries and unnecessary forwarding of subscriptions. However it incurs several drawbacks that are intrinsic to the choice of the routing algorithm but also the topology that is static and non-structured. Subscriptions are flooded to the whole network and an unsubscribe operation may implicitly unsubscribe to all the filters that are covered by the former filter. Hermes relies on an extension of Pastry [22], a structured P2P protocol named PAN. Subscriptions and publications are sent to a rendez-vous node by means of an event dissemination trees created dynamically. Notifications are forwarded by using reverse paths. More recently, BlueDove [121] proposes to match publications with subscriptions atop a modified version of Cassandra [35] in just one hop: replicating subscriptions on a selected subset of one hop away

accessible peers and then selecting one of these replica to trigger the matching according to load information of peers, regularly exchanged throughout the system. Their scheme does not require reverse path forwarding neither any computation over multiple intermediate peers.

The closest system to our is certainly Meghdoot [122]. The authors leverage the CAN [20] logical topology as we do. Similarly to other conventional publish/-subscribe systems, their data model is based on multiple attributes. Also, their system is initialized with a schema that defines attributes' name, type and domain. Additionally, all peers are assumed to share the same schema. Events are a subset of attributes from the schema set where each attribute is associated to a value. A subscription is a conjunction of binary predicates over one or more attributes. The construction of the logical space depends on the number of attributes that made up events. Assuming that events have at most n attributes, then a CAN network with $2n$ dimensions is built. Dimensions $2i$ and $2i-1$ are in charge of managing the attribute domain for attribute i. By using this property, events and subscriptions are mapped to points. Events lie on the diagonal which is a line which passes by the lower and upper bounds of the $2n$-dimensional space while subscriptions are located in the upper part delimited by the diagonal. The remaining side is left for fault tolerance purposes. To deliver notifications, each time an event is published, it is sent to the peer that manages the event point on the diagonal before triggering an event propagation algorithm that aims to reach the subscriptions affected by the event on the upper left side. The algorithm propagates the event on a selected subset of neighbors by applying some optimizations to prevent repetitive propagations. In case users subscribe for all events, their event propagation algorithm must start from a peer that is at a corner of the CAN network, which leads to performance bottleneck. Also, the main disadvantages of Meghdoot is that events type, domain (e.g. from 1 to 100 for event type integer) and the maximum number of elements per event should be defined at startup. This last parameter having a direct impact on the structured overlay and routing performances.

The main drawback from the system introduced previously and from traditional event-based systems in general is that they do not focus on the specific characteristics of RDF. The first point is that most of them make use of structured records. In a record-oriented model, events consist of named set of attribute value pairs

where the order between pairs does not matter. In contrast, RDF building blocks are 3-tuple or 4-tuple and RDF events are usually unbounded set of tuples where the order of elements within a tuple is important since the self-description of RDF comes in part from this property. In [123], the authors argue that tuples provide a simple model that is not flexible enough because subscriptions used to express events to receive must either specify for each tuple element an exact value or a wildcard (e.g. an asterisk) to point out the fact that any value matches. Record-oriented systems do not have to specify a wildcard for attributes they do not care about. They may simply omit to specify them. Although structured records seem more flexible from a subscription's perspective, the authors concede that matching by position in tuple-oriented systems is more efficient in practice. Despite the last observation regarding filtering performance, to our knowledge only one non RDF related system called JEDI [56] makes use of tuples to model events. In this system, events are spread through dissemination trees created dynamically after having elected a group leader, similarly to Hermes. However, each leader must perform a global broadcast to all other brokers, which might not be scalable depending on how it is implemented. The second drawback with traditional publish/subscribe systems is that they make some assumptions on events content in terms of type, domain and size. Unfortunately, this is not appropriate to integrate, filter and relay events that are produced from heterogeneous sources where the number of attributes and their type differ from a source to another.

To conclude, the data but also the subscription model are strongly related and have a large impact on the scenarios to consider. A few works in conventional publish/subscribe systems have based their data model on tuples. We introduce below some event-based works made for RDF.

4.1.3 RDF-based publish/subscribe systems

RDFPeers [124] is a distributed RDF repository where peers are self-organized into a Multi-Attribute Addressable Network (MAAN) [90]. MAAN extends Chord [21] such that information retrieval may be performed for any triple term. Publishing a triple implies to index it three times, each one based on the hash value of its subject, predicate and object value. Atomic, disjunctive and range subscrip-

tions are supported with the exception of some patterns. For instance, it is not possible to subscribe for all the information nor with some join constraints. Besides, RDFPeers ignores popular terms such as *rdf:type* predicates and, therefore, subscriptions involving them cannot be resolved.

In [125], Ranger *et al.* introduce an information sharing platform for disseminating RDF activities. Their solution relies on the Scribe [53] system that offers a topic-based publish/subscribe system on top of Pastry. Queries are expressed in a SPARQL dialect and registered as topics. Unlike other solutions, the algorithm they propose does not index data a priori. Instead, their strategy relies upon finding results through multicast trees built from scratch, associated with redundant caching and cached lookups mechanisms. The peers participating to the propagation are responsible for removing duplicate results within the limit of their buffer. This probably leading to duplicate notifications over the time.

CSBV [126] proposes a generic and DHT agnostic approach for resolving atomic and conjunctive SPARQL subscriptions. Their scheme strongly relies on hashing and requires to index each triple seven times. Owing to the fact that the number of indexations that is required correspond to the combination without repetition of the elements contained by the tuples that are published, it grows quickly up to 15 when quadruples are considered. Subscriptions are resolved by rewriting dynamically subscription patterns matching new incoming publications. The publish/subscribe algorithm we introduce in the next sections derives from this idea.

Recently, Shvartzshnaider *et al.* proposed in [127] to combine AI and Peer-to-Peer research approaches for building a publish/subscribe system that supports publication of arbitrary tuples and subscriptions with standing graph queries. Their idea consists in applying Rete [128] algorithms on a Chord network to resolve join conditions contained by subscriptions. Basically, a Rete network acts as a distributed cache where subscription patterns that are executed are cached along with their results for future reuse. Thus, answers from previously executed subscriptions may be reused with new subscriptions that involve similar patterns. The authors consider tuples as primitives and allow subscriptions to match against an unbounded flow of tuples but do not explain how memory growth is managed (i.e. by using for instance time windows operators). Publications and subscriptions are indexed similarly to RDFPeers (cf. Section 3.1.2) in order to create rendezvous

nodes where the satisfaction of subscriptions is verified. Although they claim that Rete approach is effective, no discussion is given about how duplicates are avoided when in-memory buffers overflow. Moreover, subscriptions are formulated through an ad-hoc scripting language and no experimental evaluation is available.

Summary

Although publish/subscribe is generating great interest these last years, the model is not new. Many approaches have been proposed, especially in a distributed context. Most of these systems made assumptions on events type, domain and size to improve the overall performances. A lot are using hashing to balance data on multiple nodes, thus implying to index the same data multiple times while reducing range-queries efficiency. Finally, a few, not to say none, try to combine filtering and persistent storage of RDF events for later analysis. The solution we introduce in the next section aims to tackle these drawbacks.

4.2 Publish/Subscribe Infrastructure for RDF

The publish/subscribe infrastructure for RDF events we describe here is built atop the infrastructure introduced in the previous chapter and as a consequence it reuses some concepts and notions. The purpose of this extension is to offer the possibility to react to data in a responsive manner and gradually, when events are coming. In the following, we describe how events are modeled and how subscriptions are expressed through our data and subscription model. Then, we present the different requirements we place on our infrastructure in order to better explain the different choices we made with the two and complementary event filtering algorithms we propose.

4.2.1 Data and subscription model

The main task of distributed publish/subscribe systems is to relay data to interested parties. Information to disseminate but also interest to data are described by an event model. Data that are published are events and interest in events

is formulated by means of subscriptions. When the time comes to decide how events and subscriptions are represented it is really important to keep in mind the compromise there is between expressiveness and scalability [129]. To summarize, the more expressive a publish/subscribe system is, the more complex the event filtering algorithm becomes. The direct consequence is that the efficiency of the matching algorithm significantly affects both performance and scalability. A second critical point to consider is about interoperability. In distributed systems the ability to automatically interpret the information exchanged meaningfully and accurately between heterogeneous machines is often a strong prerequisite. In our case it is all the more true since we have as a requirement to store events for future reuse with standard technologies from the Semantic Web stack. The publish/subscribe messaging pattern already provides a method to enhance interoperability by decoupling publishers and subscribers. However, the event model plays also an important role. Consequently, to further improve interoperability, it is a good practice to use an event model that reuses or extends open standards that are platform and language independent. Our data and subscription model strive to address these two challenges by reusing existing standards and by defining clear limits about which kind of interest may be formulated with the filtering language we have adopted. Our approach, described in [130], allows users to formulate queries and subscriptions but also to insert and publish information with respectively an extension of RDF and a subset of SPARQL.

Events

In our data model, events are occurrences or actions of something that happens expressed in the RDF model using 4-tuples (i.e. quadruples) whose elements are named RDF terms. An RDF term may be either an IRI or a Literal value. Blank nodes are not allowed because they incur expensive mechanisms to ensure their uniqueness in the whole system. In compensation, end-users publishing events containing bnodes may use skolemization[1] to transform bnodes to IRIs.

Regarding the granularity of events, a quadruple has a limited meaning since it can embed only a few number of information. It is acknowledged that fine

[1] `http://www.w3.org/TR/rdf11-concepts/#section-skolemization`

grain events significantly complicate the programming process and reduce the performance of the system [131]. We tackle this issue by introducing Compound Events (CEs). Elements or quadruples generated at the same time by a given source form a CE, as defined by (4.1b). Each CE is a list of quadruples where quadruples share a common term called graph value. This term is built with a combination of a unique source identifier and a timestamp. The purpose of this graph value is threefold. It is used to identify the event source, the event itself but also to offer the possibility to link together several quadruples for emulating unbounded multi-attribute values like in conventional publish/subscribe systems.

$$q = (g, s, p, o) \mid g, s, p, o \in RDFTerm \tag{4.1a}$$

$$CE = (q_1, ..., q_i, ..., q_n) \mid q_i = (g, s_i, p_i, o_i) \tag{4.1b}$$

Events indexation rests upon the routing algorithms proposed in the previous chapter. The Figure 4.1 shows how CEs are mapped to our revised CAN network. In concrete terms, each quadruple from a CE is indexed independently. A specific message is created per quadruple with a unicast key based on the quadruple's terms. Then, this message is routed until reaching the peer that manages the quadruple's terms. It is worth to notice here that quadruple and thus Compound Event indexation is fully asynchronous. Each method invocation done indirectly by a publisher to index a quadruple during a Compound Event publication returns immediately and no response is sent back. Therefore, quadruples from a CE are inherently indexed in parallel.

Subscriptions

Subscriptions aim to match CEs. They are content-based and formulated using a subset of SPARQL. In essence, a subscription is basically a list of one or more atomic and range queries called sub-subscriptions or SS. A subscription is applied on different CEs independently, i.e. only the quadruples that belong to the same CE can trigger a notification. More precisely, a subscription $S = (SS_1, SS_2, ..., SS_n)$ is found to match a compound event $CE = (q_1, q_2, ..., q_m)$ if for each SS_i there exists at least a matching q_j. In other words, the whole subscription should be satisfied

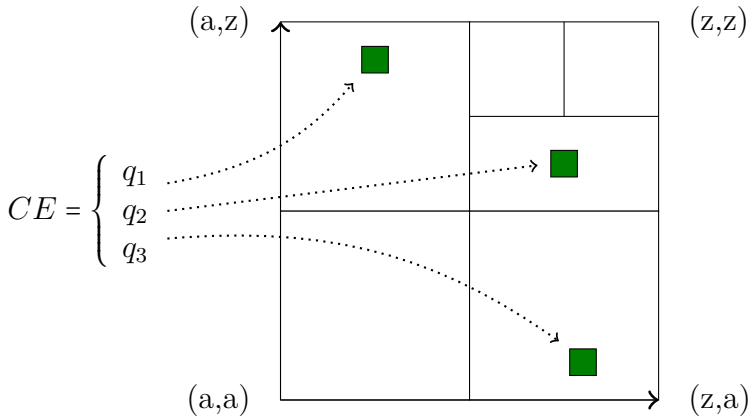


Figure 4.1 – Compound Event distribution with three quadruples.

by a subset of the quadruples contained by a CE. Although SPARQL could be used to formulate more complex subscriptions with filtering that involves patterns made of multiple events, a domain usually known as composite event detection [132, 133] and supported by CEP engines, we limit our system to simple event processing. The reason lies in the fact that composite event detection enhances the expressiveness of event-based systems but at the price of expensive correlations to perform in a distributed manner, thus slowing down the performance of the whole system. Moreover, many applications do not need complex subscriptions. However, as we will see in Chapter 6 our solution is flexible enough to be extended or combined with additional tools (like we did in the PLAY EU project) for complex event processing.

To better illustrate which kind of subscriptions we allow, let suppose a building with a sensor that generates events with our data model each time a person enters or exists from the front door. Assuming the action, the person name and age is embedded by the CEs that are published, a possible subscription to get informed about the name of all people who have 25 years old or more and that exit from the front door is depicted by Listing 4.1. This example is a standard SPARQL query that could be executed synchronously on a common RDF engine. In our case it can be seen as a query over future events, or say in another way, as a long standing query. However, SPARQL is a really expressive language and since we want to ensure expressiveness while maintaining scalability, but also due to the

different requirements we introduce in the next section, we restrict subscriptions to a subset of SPARQL with some conditions. A subscription written with our subscription model is assumed to be a SPARQL query that *a*) uses the SELECT query form; *b*) contains at most one group GRAPH pattern with a graph variable; *c*) returns the graph variable declared in the GRAPH pattern. Multiple triple patterns may be used inside the graph pattern defined in the subscription. One or more FILTER clauses are also allowed to restrict solutions. Standard logical operators but also filter functions like REGEX, STRSTARTS, etc. are permitted.

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 PREFIX ex: <http://example.org/v/>
3 SELECT ?g ?id ?name WHERE {
4     GRAPH ?g {
5         ?id ex:action "exits" .
6         ?id foaf:name ?name .
7         ?id foaf:age ?age
8     } FILTER (?age > 25)
9 }
```

Listing 4.1 – SPARQL subscription example.

The process to index and detect subscriptions that are satisfied by incoming events depends on the event filtering algorithm considered. Thus, the scheme will be explained while describing the proposed publish/subscribe algorithms. However, the basic scheme to decompose subscriptions before indexing them remains the same whatever the event filtering algorithm is. It consists in extracting atomic and range queries, named in this context sub-subscriptions, from the subscription in order to have the possibility to send the subscription to the peers that manage one or more of the extracted sub-subscriptions. Specifically, the subscription given in Listing 4.1 results in a decomposition with three SSs: two atomic queries (one on line 5 and one on line 6) and one range query (lines 7–8).

To give a brief idea of how subscriptions are mapped on a CAN network, Figure 4.2 outlines a simple case where two subscriptions made of one SS each are indexed on a 2D CAN network. In this specific case, the general rule to index a subscription consists of sending the subscription to the peers responsible for the fixed parts of the SS it embeds. Since an SS may contain free variables, it may

reach multiple peers. Moreover, some subscriptions may overlap on a peer.

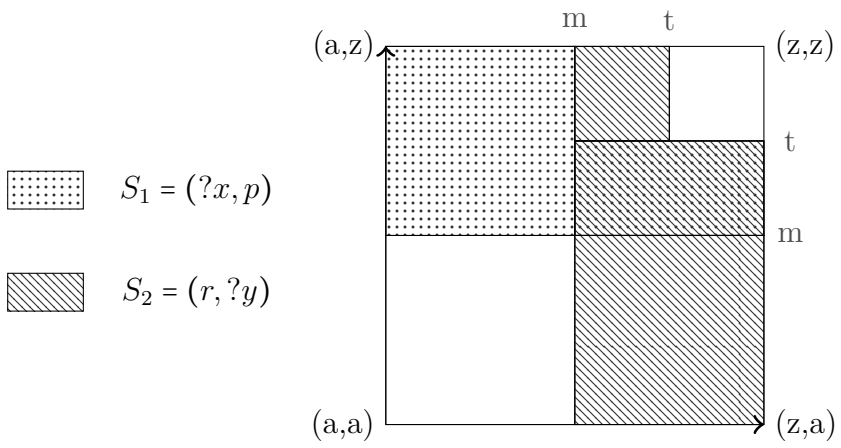


Figure 4.2 – Distribution of two subscriptions overlapping on a peer.

4.2.2 Requirements

In addition to the data and subscription model our publish/subscribe infrastructure complies with, the system is also designed to enforce a set of properties. We list below the different properties or requirements we decided to support and that have, along with the models, a direct impact on how the publish/subscribe algorithms described in the next section behave. Some are use case driven.

R1 Events and subscriptions are assumed to be submitted to the event notification service by means of lightweight applications dubbed proxies which represent clients of the brokering network. Publish and subscribe proxies are distinguished. The former is used to publish events whereas the latter forwards subscriptions to the P2P network in charge to perform the matching between events and subscriptions. The general purpose of proxies is to let the possibility to enforce end-to-end properties that have not been or could not be implemented at the P2P level.

R2 Clocks are assumed synchronized between machines inside the P2P network with Network Time Protocol (NTP) but not between proxies that subscribe and publish events to peers. Indeed, to synchronize all entities is not an

acceptable assumption due to the extra overhead latency incurred by far away machines on the Internet. However, to synchronize peers' clocks when peers deployment targets a closed environment like a cloud, a cluster or a grid as it is in our case, remains an acceptable possibility.

R3 Causal ordering between publish and subscribe requests handled asynchronously from a same proxy must be enforced to do not introduce false negative (i.e. notifications which are not received whereas they should). However, we do not want to enforce any delivery order between publications and/or subscriptions from different proxies since the order may only be guaranteed in that case if events are tagged with a timestamp from a global clock shared between all entities and if the communication network provides a guaranteed fixed latency time [134]. To illustrate the issue, let consider a subscription S which enters the P2P network and is timestamped before a publication P but eventually P is received before S on the peer that performs the matching. Such a scenario can occur due to the asynchronism of the operations and the multiple hop communications between peers. Moreover, this issue arises even if we use TCP as the underlying protocol since the connection guarantees delivery order between two entities only (e.g., peers, proxies-to-peer, peer-to-user).

R4 Quadruples must eventually be stored in the P2P overlay on peers in order to be retrieved later to compute expensive batch analysis, statistical information or simply to help CEP engines to correlate different CEs [12] in a distributed manner by leveraging past knowledge.

R5 Events that must be notified are notified. For instance, if we publish P_1 and P_2 from a same publisher and both are matching S, then the subscriber who has subscribed with S will receive P_1 and P_2. However, if due to the asynchronism P_2 is notified before P_1 we can detect the situation and report the issue to the interested parties. Here, we do not focus on the fault tolerance aspect and we let it for future works. Consequently, events and subscriptions may be lost if some failures occur.

R6 Data indexation does not rely on hashing in order to avoid multiple index-

ations of the same publications as it is the case for most of the existing works. Instead, publications should be indexed according to their lexicographic order. On the other side subscriptions are replicated on multiple peers in accordance with their expressiveness. We strongly believe that a real publish/subscribe system face more publications than subscriptions.

R7 Notifications aim to inform subscribers about events that satisfy their subscription(s). We think that subscribers should have the possibility to receive different types of notifications that carry more or less information about the events that match subscriptions. The purpose is to enhance the delivery time but also the end-user bandwidth consumption when full event values are not required. Consequently, we assume a subscriber is allowed to subscribe for getting either a signal, a collection of bindings (the values matching the variables contained by the subscription) or the full compound event that has matched its interests. This is materialized with a subscription by using respectively Bindings, Compound Event or Signal notification listeners.

R8 Our event notification service is assumed to deliver no duplicate and no false positive. By false positive we mean to deliver events that are not matching a subscription.

4.2.3 Event filtering algorithms

This section introduces two publish/subscribe algorithms optimized for different use cases. The first one, named Chained Semantic Matching Algorithm (CSMA), is optimized for the publications while One-step Semantic Matching Algorithm (OSMA), introduced page 109, is optimized for the subscriptions.

CSMA

The general idea of CSMA, as inspired by Liarou *et al.* [126] with Continuous Spread By Value (CSBV), is to publish in parallel and perform the matching of all the sub-subscriptions contained by a subscription sequentially. Indeed, all peers involved in a subscription will be organized in a chain-like fashion. Only the peers indexing the subscription can start the matching process and notify the next peers

in the chain which, in turn, will try to find a match. The process ends when reaching the last peers in the chain, i.e. when the whole subscription is satisfied. We decided to start from CSBV because this algorithm is designed to balance quadruples among peers by handling them independently and not considering a CE as a whole.

Indexing a publication Compound events published by users to the network are sent first to a publish proxy. The proxy relays the compound event to a peer that timestamps the quadruples contained by the CE with the peer's clock. This is done on a peer and not from a proxy to satisfy R2 and R3. If the quadruples were timestamped on the proxy side, a publication time could not be compared with a subscription time once a publication reaches a peer because proxies and peers do not have their clocks synchronized. Once timestamps are set, the first peer that receives the compound event considers each quadruple from the CE independently and sends each of them, asynchronously, to the responsible peer as depicted by Listing 4.2. Notice that quadruples are sent along with an additional one, named meta-quadruple. This quadruple denotes the number of quadruples contained by a compound event and is used later to retrieve the entire compound event that has matched a subscription in case the full CE was requested (requirement R7). Each quadruple is sent asynchronously from a publish proxy to the event service with the help of a specific request message that is then routed to the relevant peer with the algorithm introduced in Section 3.2.2 since the request embeds a unicast key where elements are RDF terms from the quadruple to index.

```
1  def receive(compound_event):
2    timestamp = now()
3
4    for quadruple in compound_event:
5      quadruple.indexation_time = timestamp
6      async_send(PublishQuadrupleRequest(quadruple))
7
8    # creates a meta-quadruple that indicates the number
9    # of quadruples contained by the compound event
10   meta_quad = create_meta_quad(compound_event, timestamp)
11   async_send(PublishQuadrupleRequest(meta_quad))
```

Listing 4.2 – Upon reception of a publication on a peer.

Then, when the peer that manages the quadruple terms receives the request that aims to index the quadruple, it triggers the matching algorithm defined in Listing 4.3.

Quadruple matching The purpose of the matching algorithm is twofold. First, it stores the received quadruple q to ensure that delayed subscriptions (i.e. subscriptions injected in the network before the quadruple but indexed after due to different routing steps) have a chance to be fulfilled. Second, it detects the subscriptions that are fully or partially matched in case the quadruple received is not a meta-quadruple added by the system. For each subscription S which is detected to be matched by the new quadruple, the number of sub-subscriptions contained by S is checked. To have S with more than one sub-subscription A means that only one part of the whole subscription is matched. When this situation occurs, S is rewritten into S' as in CSBV. The rewrite operation consists of creating a new subscription that does not contain the sub-subscription A which is verified, and to replace in the remaining sub-subscriptions for S' the variables from A with the values from q. Afterwards, S' is indexed, again, as for S, by considering the first sub-subscription contained within S'. One additional step has to be considered during the rewrite operation when the subscription that is analyzed was made with a Bindings notification listener. Such a listener implies to send back to the users only the bindings that are matching the subscription. However, we do not want to convey the intermediate results from peers to peers when the subscription is rewritten because the size of a literal associated to a quadruple that has potentially matched S is not bounded: it could be some bytes or megabytes. For this purpose, intermediate results are stored on the peer matching the SS and a reference to that peer along with a hash value identifying the intermediate results is added as metadata (by using a 128 bits non-cryptographic hash function) to the rewritten subscription. The hash value allows us to carry an identifier with a small and predefined size. Moreover, even if the identifier is unique with a high probability, a sequence number may be concatenated to guarantee the uniqueness.

```
1  def receive(publish_request):
2    quadruple = publish_request.quadruple
3    store(quadruple)
4
5    if not is_meta_quad(quadruple):
6      # finds subscriptions that have their first sub-subscription
7      # matched by the quadruple received
8      subscriptions = find_subscriptions_matching(quadruple)
9      for subscription in subscriptions:
10       if quadruple.indexation_time >=
11           subscription.indexation_time:
12         rewrite_or_notify_subscriber(subscription, quadruple)
13
14 def rewrite_or_notify_subscriber(subscription, quadruple):
15   if len(subscription) == 1:
16     notify_subscriber(subscriber, quadruple)
17   else:
18     rewrite_and_index(subscription, quadruple)
19
20 def rewrite_and_index(subscription, quadruple):
21   if subscription.listener_type is BINDINGS:
22     intermediate_result =
23       create_intermediate_result(subscription, quadruple)
24     store(intermediate_result)
25
26   async_send(IndexSubscriptionRequest(rewrite(subscription)))
27
28 def notify_subscriber(subscription, quadruple):
29   # filter the quadruple according to the subscription
30   # type and its variables
31   chunk = filter(quadruple, subscription)
32   async_send(subscription.proxy_url,
33       NotifyRequest(subscription.id, chunk))
34
35   if subscription.listener_type is BINDINGS:
36     # contact peers that have stored intermediate results
37     for peer, hash in subscription.intermediate_peers:
38       async_send(peer,
39           CollectIntermediateResultsRequest(
```

```
40                  subscription.id, hash))
```

Listing 4.3 – Upon reception of a PublishQuadrupleRequest by a peer.

In case the number of sub-subscriptions contained by S is one, no more quadruple are necessary to match S. The only thing to do is to notify the subscriber about a solution by using the right format in line with the original notification listener type. According to the type of subscription listener used by the subscriber we trigger different kind of notifications.

In case of a *Signal* or *Compound Event* (more detail follow for this last case) listener, we send back a pair made of the subscription identifier plus the graph value of the last quadruple that has matched the subscription. However, when a Bindings listener is used, we replace the graph value by the values associated to the variables matching the last SS. But, again, to trigger a notification for Bindings implies an additional operation which involves the different peers that store the intermediate results. These peers have to be contacted and asked to return, in parallel, to the subscriber the missing parts of the subscription that have been matched. Finally, when the different parts of Bindings are collected, they are merged and the result is passed to the listener before being executed (cf. Listing 4.4). The delivery is performed when the subscriber has received a number of solutions equals to the number of result variables contained by the initial subscription. The Signal case is handled immediately by executing the associated listener.

To handle *Compound Event* notification listeners is a bit more complicated. Previously, we said that when a subscription is fully matched we send back to the subscribe proxy the graph value only, as for notification listeners of type *Signal*. This graph value g is used to query synchronously all the peers that satisfy the quadruple pattern $(g, ?s, ?p, ?o)$. This operation is repeated periodically until to receive all the quadruples that made up the CE matching the subscription. Indeed, a simple query is not sufficient because all the quadruples contained by the compound event may not be indexed when the last sub-subscription is satisfied and in this case, notice that these quadruples were not necessary for the matching subscription process to succeed. Moreover, we decided not to send quadruples to the subscriber as soon as they are matching sub-subscriptions for two reasons.

First, only the first SS but not all may be satisfied. In such a case, the intermediate results are kept by the subscriber and overload it until an unsubscribe operation or a garbage collection is performed. The latter potentially preventing some events to be delivered, thus going against requirement R5. The second reason is that the number of quadruples matching a subscription may be just a subset of the full CE. Thus, even if intermediate matching quadruples were sent gradually, it would be necessary to perform an extra step later to retrieve the remaining chunks.

To avoid sending back to the subscribe proxy quadruples that may have already been received during a previous polling operation, quadruples' position (index) from their respective CE list are attached to the *ReconstructCompoundEventRequest* which is sent. Thanks to this information, only new expected quadruples are sent back and some bandwidth is saved. The polling period could also be tuned based on an exponential backoff [135] or inversely proportional to the times returned by such a function but it is scenario sensitive and specific.

```
1  def receive(notification):
2    subscription = find_subscription(notification.subscription_id)
3    listener = find_listener(notification.subscription_id)
4    listener_type = type(listener)
5
6    if listener_type is BINDINGS:
7      if get_nb_chunks_received(notification.id) ==
8          subscription.nb_result_vars:
9        chunks = remove_and_merge_chunks(notification.id)
10       if mark_as_delivered(notification.id):
11         listener.deliver(subscription.id, chunks)
12     else:
13       memorize_chunk(notification.id, notification.chunk)
14   elif listener_type is SIGNAL:
15     graph_value = notification.chunk
16
17     # returns False if notification.id already delivered
18     # notification id unique for a given subscription and CE
19     if mark_as_delivered(notification.id):
20       listener.deliver(subscription.id, graph_value)
21   elif listener_type is COMPOUND_EVENT:
22     ce = reconstruct_compound_event(notification)
```

```
23      if ce is not None:
24        listener.deliver(subscription.id,
25            CompoundEvent(quadruples_received))
26
27  def reconstruct_compound_event(notification):
28    expected_nb_quadruples = -1
29    quadruples_received = set()
30
31    if not mark_as_delivered(notification.id):
32      return None
33
34    while expected_nb_quadruples == -1
35        or not len(quadruples_received) == expected_nb_quadruples:
36      graph_value = notification.graph_value
37
38      response = sync_send(
39          ReconstructCompoundEventRequest(graph_value),
40              indexes(quadruples_received))
41
42      for quadruple in response.new_quadruples:
43        if is_meta_quad(quadruple):
44          expected_nb_quadruples = get_meta_quad_value(quadruple)
45        else:
46          quadruples_received.add(response.new_quadruples)
47
48      if not len(quadruples_received) == expected_nb_quadruples:
49        sleep(TIMEOUT)
50
51    return CompoundEvent(quadruples_received)
```

Listing 4.4 – Upon reception of a notification by a subscribe proxy.

Listing 4.4 contains some particular conditions on line 10, 19 and 31 that rely on a *Compare-and-Swap* operation to prevent duplicates to be delivered (requirement R8). Indeed, this first algorithm may suffer from duplicate notifications when an *accept all* subscription (i.e. a subscription that matches any event) is handled or a CE with objects list (i.e. a CE that contains two or more quadruples that share

the same graph, subject and predicate but different object values) are published[2]. This usually happens when there are more than one quadruple from a CE that satisfy a same sub-subscription. In other words, the issue occurs when we have a subscription S with n SSs, a CE with q quadruples whose m are matching S and $m > n$. When such a case occurs, up to m notifications maybe triggered to the subscribe proxy whereas only one is expected per CE in case the subscriber subscribes for example with a *Signal* listener. Also, the condition on line 31 is essential to avoid several reconstructions for a same CE since the `receive` and hence the `reconstruct_compound_event` operations may be triggered in parallel by different peers on a subscribe proxy for a same CE due to duplicates.

Indexing a subscription Initially, a subscription is submitted from a user through a subscribe proxy. As for a publication, a subscription is relayed from a proxy to a peer which is responsible for timestamping and indexing it. Listing 4.6 summarizes the process which is triggered once a subscription is received by a proxy. First the SPARQL query is decomposed into atomic or range queries. Listing 4.5 shows an example of the expected pieces to consider for indexing the subscription from Listing 4.1 once it has been decomposed. The atomic and range queries that result from the decomposition are the smallest request entities that may be routed by taking advantage of the overlay structure.

```
1 SS_1 = (?g, ?id, ex:action, "exits")
2 SS_2 = (?g, ?id, foaf:name, ?name)
3 SS_3 = (?g, ?id, foaf:age, ?age) FILTER (?age > 25)
```

Listing 4.5 – SPARQL subscription decomposition into sub-subscriptions.

After the split of the subscription into pieces, a unique identifier is generated and the subscription is conveyed to the peer the proxy is aware of as an overlay entry point. The first peer that receives the subscription sets the indexation timestamp and routes the subscription asynchronously on the network to all the peers

[2]The simplest scenario that leads to duplicate notifications is the one involving two peers which register an *accept all* subscription and a publisher that publishes a CE with 2 quadruples that each reaches a different peer. Since the subscription is registered on both peers and quadruples from a same CE match on two different peers, two notifications are sent towards the subscriber, which causes duplicates if there are not filtered.

that match the first sub-subscription. The subscription will be further distributed among the peers in a chain like fashion. To use the first sub-subscription ensures that the subscription is indexed at least on a peer that will receive later any publication that can match this first SS. We also pay attention to use only the SS to avoid some extra synchronization points, and thus communications compared to the case where the subscription would be indexed on all the peers matching any of the SSs. Once decomposed, a subscription is represented by its atomic and range queries and these share at least one common variable: the graph variable that represents the CE identifier which is matching the subscription. When common variables are shared between atomic or range queries, they have to be resolved as an equi-join. Choosing to resolve sub-subscriptions in parallel would imply a consensus between some peers due to equi-joins to compute and we think that this agreement between peers is more expensive (or at least not interesting for the few number of atomic or range queries a subscription usually embeds) than the chain-like approach where the synchronization and agreement is implicit.

```
1  # subscribe on the proxy side
2  def subscribe(sparql_query, listener):
3    subsubscriptions = decompose(sparql_query)
4
5    # the subscribe proxy is remotely accessible to
6    # receive notifications
7    proxy_url = get_subscribe_proxy_url()
8
9    subscription_id =
10       hash(sparql_query, datetime.now(), proxy_url) + proxy_id
11   subscription =
12       Subscription(subscription_id, subsubscriptions,
13           proxy_url, type(listener))
14
15   memorize(subscription_id, (subscription, listener))
16   async_send(peer, SubscribeRequest(subscription))
17   return subscription_id
18
19 # subscription received on a peer
20 def subscribe(subscribe_request):
```

```
21    subscribe_request.subscription.indexation_time = now()
22    # index subscription to all peers matching subsubscriptions[0]
23    async_send(peer,
24        IndexSubscriptionRequest(subscribe_request.subscription))
```

Listing 4.6 – Handling a subscription from a proxy to a peer.

Subscription matching Once a peer receives an *IndexSubscriptionRequest*, it triggers the matching algorithm defined in Listing 4.7. It is really similar and symmetric to the behavior applied for indexing a quadruple. First, the subscription is stored in the local semantic datastore to ensure that future quadruples matching the subscription have a chance to be detected. Then, a SPARQL query is built on the fly from the subscription. This query is used to retrieve the quadruples that are matching the subscription. At this step, to find some quadruples matching the subscription implies that the subscription itself has been delayed regarding the original timestamps-based order in which it has entered the network. Finally, for each quadruple matching the subscription we apply the method `rewrite_or_notify` subscriber. The behavior of this method is exactly the same as for indexing a quadruple; it either rewrites the subscription into a new one if some sub-subscriptions still have to be satisfied or it notifies the subscriber about a solution.

```
1  def receive(index_subscription_request):
2    subscription = index_subscription_request.subscription
3    store(subscription)
4
5    # query the local semantic datastore
6    quadruples_matching = find_quadruples_matching(subscription)
7
8    for quadruple in quadruples_matching:
9      if quadruple.indexation_time >=
10         subscription.indexation_time:
11       # see Listing 4.3 for details about the next call
12       rewrite_or_notify_subscriber(subscription, quadruple)
```

Listing 4.7 – Upon reception of a IndexSubscriptionRequest on a peer.

To summarize, the subscription is distributed among the peers in a chain like fashion. Thus, the sub-subscriptions contained by the subscription are handled step by step in the order they appear. A subscription is stored on a peer found using the fixed parts of the first atomic query, so potentially many peers. Each peer stores the whole subscription to be able to find the next peers to reach when a sub-subscription is verified. The peers that store the first sub-subscription are the head of the chain. This algorithm is designed to transparently handle the common case and the temporal ordering discrepancy, we refer to as the happen-before relation. As an improvement, sub-subscriptions could be shook up according to the number of fixed parts they contain. The peers could also maintain and exchange statistical information about the most frequently met sub-subscriptions. Thanks to this information we could reorganize SSs embedded within a subscription such that the less frequently seen SSs are used in first. It should improve the load-balancing and the time to detect non-matching events but not those that fully match a whole subscription. Therefore, this optimization is left for future work.

Avoiding polling for reconstruction

The first version of the publish/subscribe algorithm we explained has a main shortcoming. The CE listener uses polling for retrieving the quadruples which compose a CE that satisfies a subscription. Fast polling wastes resources on peers whereas slow polling increases the delivery time. The first and obvious improvement is to avoid polling for users that wish to receive the full CEs content (i.e. they have subscribed with a notification listener of type Compound Event). To carry out this goal, we propose to use a push mechanism from peers. The push procedure relies on another type of subscriptions called *ephemeral subscriptions*. The idea is to send gradually to the subscribers the multiple quadruples that made up a CE, this without having to poll periodically from a subscribe proxy to the P2P network for missing chunks.

To achieve this purpose, the method `notify_subscriber` introduced in Listing 4.3, and some others, have to be edited as follows. Once a subscription S is matched, an *ephemeral subscription* is indexed on the right peers. These peers will asynchronously and lazily notify the client proxy for the different quadruples that

compose the CE matching S as soon as they arrive. Listing 4.8 summarizes the basic idea of the algorithm with new parts which are highlighted.

```
1  # called by any peer that has a subscription matched by an event
2  def notify_subscriber(subscription, quadruple):
3    # When a COMPOUND_EVENT listener is used, the chunk is now
4    # a quadruple and not the event identifier
5    chunk = filter(quadruple, subscription)
6    async_send(subscription.proxy_url,
7        NotifyRequest(subscription.id, chunk))
8
9    if subscription.listener_type is BINDINGS:
10     # contact the peers that have stored the intermediate
11     # results identified by a hash value
12     for peer, hash in subscription.intermediate_peers:
13       async_send(peer,
14         CollectIntermediateResultsRequest(subscription.id, hash))
15   elif subscription.listener_type is COMPOUND_EVENT:
16     async_send(
17         IndexEphemeralSubscriptionRequest(quadruple.graph))
18
19 # handle performed on the reception of an ephemeral subscription
20 # on a peer managing it
21 def receive(ephemeral_subscribe_request):
22   store(ephemeral_subscribe_request)
23
24   quadruples =
25       find_quadruples_matching(ephemeral_subscribe_request)
26   # we may have several quadruples indexed on the same peer
27   # that belong to the same CE
28   for quadruple in quadruples:
29     async_send(ephemeral_subscribe_request.proxy_url,
30         NotifyRequest(ephemeral_subscribe_request.id, quadruple))
31
32 # executed by a peer indexing a quadruple
33 def receive(publish_request):
34   quadruple = publish_request.quadruple
35   store(quadruple)
36
```

```
37    subscriptions = find_subscriptions_matching(quadruple)
38    for subscription in subscriptions:
39      if quadruple.indexation_time >= subscription.indexation_time:
40        rewrite_or_notify_subscriber(subscription, quadruple)
41
42    ephemeral_subscriptions =
43        find_ephemeral_subscriptions_matching(quadruple)
44    for ephemeral_subscription in ephemeral_subscriptions:
45      async_send(ephemeral_subscription.proxy_url,
46          NotifyRequest(ephemeral_subscription.id, quadruple))
47
48  # method invoked when a notification that embeds a chunk
49  # (i.e. a quadruple or bindings) is received by a subscribe proxy
50  def receive(notification):
51    subscription = find_subscription(notification.subscription_id)
52    listener = find_listener(notification.subscription_id)
53    listener_type = type(listener)
54
55    if listener_type is BINDINGS:
56      if get_nb_chunks_received(notification.id) ==
57          subscription.nb_result_vars:
58        chunks = remove_and_merge_chunks(notification.id)
59        if mark_as_delivered(notification.id):
60          listener.deliver(subscription.id, chunks)
61        else:
62          memorize_chunk(notification.id, notification.chunk)
63    elif listener_type is SIGNAL:
64      graph_value = notification.chunk
65
66      # returns False if notification.id already delivered
67      # notification id unique for a given subscription and CE
68      if mark_as_delivered(notification.id):
69        listener.deliver(subscription.id, graph_value)
70    elif listener_type is COMPOUND_EVENT:
71      quadruple = notification.chunk
72      if not already_delivered(notification.id):
73        # nb_quads_expected is initialized to -1 if not defined
74        # a pair made of two values is retrieved
75        quads, nb_quads_expected =
```

```
76              tmp_store.get_chunks(notification.id)
77
78          if is_meta(quadruple):
79            nb_quads_expected = extract_meta_value(quadruple)
80          else:
81            quads = quads.add(quadruple)
82
83          if len(quads) == nb_quads_expected and
84              mark_as_delivered(notification.id):
85            listener.deliver(subscription.id, CompoundEvent(quads))
86            tmp_store.remove_chunks(notification.id)
87            async_send(RemoveEphemeralSubscriptionRequest(
88                quadruple.graph))
89          else:
90            tmp_store.add_or_update_chunks(notification.id,
91                (quads, nb_quads_expected))
```

Listing 4.8 – Pushing compound events to subscribers. New lines or edited parts, compared to the solution based on polling, are highlighted.

The general behavior of the algorithm remains the same as the one that uses pushing. A subscription is indexed on several peers and rewritten each time a new quadruple satisfies it. However, once a peer detects a subscription which can no more be rewritten, due to the number of sub-subscription which is equal to one, the peer sends back to the subscriber the chunk that has fulfilled part of the subscription (i.e. the quadruple that has matched the last sub-subscription embed by the subscription). In addition, an *IndexEphemeralSubscriptionRequest* is sent to all the peers managing the CE identifier (which is equals to the graph value of any one of the quadruples contained by the CE). Thanks to ephemeral subscriptions, the quadruples that are part of a CE matching a subscription, but not involved in the matching, can be tracked down. The detection is done when an ephemeral subscription is indexed and when a peer indexes a new quadruple. Both sides are required to guarantee correct delivery against delayed packets, as already explained for the default subscriptions.

The last update affects the proxy which is in charge of receiving the notifications. Each time it receives a chunk, which corresponds to a quadruple, it checks whether the quadruple is a meta-quadruple or not. The meta-quadruple is, as we

already explained, a quadruple that is automatically added to a CE when it is created. This particular quadruple denotes, the number of quadruples contained by the compound event the proxy is in charge of regenerating. Thus, when a quadruple is analyzed as a meta-quadruple, its object value connotes the number of quadruples which is expected before delivering the CE. While the number of intermediate chunks received is different of the expected number of quadruples, the quadruples are stored in a temporary datastore. Otherwise, a new CE is delivered after having reconstructed it from the multiple chunks that are then removed from the temporary datastore. Finally, a message is sent asynchronously to all the peers that were indexing the ephemeral subscription in order to remove it. This request should be supplemented by a periodic garbage collection operation, performed on each peer, to remove the ephemeral subscriptions that should have expired. This ensures that no memory or disk leak occurs due to remove requests that may never reach the peers if a brutal failure occurs on a proxy.

This second version avoids polling but has a cost, that of maintaining more states per peer and performing new actions on the reception of a quadruple, a subscription or an ephemeral subscription. Also, an ephemeral subscription must not be removed before a subscriber has received and delivered the associated CE. Thus, to decrease the search space at each ephemeral matching operation, a garbage collection is required, which is not free.

Polling vs Pushing We are now comparing polling and pushing to see what could be the tradeoffs, especially according to the subscription and CE size. A common case where we don't have subscription of type *accept all* neither CE with objects list nor ordering issue is assumed. Also, method calls can be made in parallel. To compare the two solutions we identified the following parameters:

- l, the subscription chain length (number of SS);
- t_{ci}, the average time to route a request inside the P2P network;
- t_{co}, the average time to forward a request from a proxy to a node in the P2P network or from a peer to the outside;
- t_m, the average time to match a subscription or an ephemeral subscription;

- q, the CE size (i.e. number of quadruples);
- x, the number of poll actions required to reconstruct a full CE;
- t_p, the polling period.

Both methods differ on how a CE that satisfies a subscription is reconstructed and delivered to the corresponding subscriber. However, they perform the same steps to detect whether a CE satisfies a subscription. A rough estimation of the time required to perform the common steps, referred to as $T_{detect\_matching}$, is given in first. This time can be simplified as the sum of the time required to index a subscription, index a CE and rewrite the initial subscription until to have the last SS that matches a quadruple from the CE which has been indexed.

$$T_{detect\_matching} = T_{index\_subscription} + T_{index\_ce} + T_{rewrite\_subscription} \tag{4.2}$$

Indexing a subscription or a compound event requires to forward a payload from a proxy to a peer before a request is routed inside the P2P network, thus requiring $t_{co} + t_{ci}$ each. Although a CE contains q quadruples, only t_{ci} is required because quadruples are dispatched in parallel. Once indexation is done, $l - 1$ rewriting steps are triggered by assuming the CE satisfies the original subscription (not all quadruples from a CE is matching the subscription). Therefore, the rewriting operations requires $(l-1) * (t_m + t_{ci}) + t_m$.

$$T_{index\_subscription} = t_{co} + t_{ci} = T_{index\_ce} \tag{4.3}$$

$$T_{rewrite\_subscription} = (l-1) * (t_m + t_{ci}) + t_m \tag{4.4}$$

$$T_{detect\_matching} = 2 * t_{co} + l * (t_m + t_{ci}) + t_{ci} \tag{4.5}$$

Hence, a rough estimation of $T_{detect\_matching}$ is $2 * t_{co} + l * (t_m + t_{ci}) + t_{ci}$. Now that the time to detect a matching is known, we start to assess the overall time required to deliver an event with respectively polling and pushing. Again, the times are decomposed as the sum of simpler ones for the sake of the explanations.

$$T_{delivery\_polling} = T_{detect\_matching} + T_{notify\_ce\_id} + T_{reconstruct\_polling} \tag{4.6}$$

$$\begin{aligned} T_{delivery\_pushing} = {} & T_{detect\_matching} + T_{notify\_subscriber} + \\ & T_{index\_ephemeral\_subscription} + T_{handle\_ephemeral\_subscription} \end{aligned} \tag{4.7}$$

Let's start with $T_{delivery\_polling}$. The first time $T_{detect\_matching}$ is the one described above while $T_{notify\_ce\_id}$ corresponds to the time that is consumed to forward the graph value to the subscribe proxy in one hop, which is t_{co}. Finally, for $T_{reconstruct\_polling}$, $x * (t_{co} + t_{ci} + t_p) - t_p$ is needed for sending the request and $x * (t_{ci} + t_{co})$ for routing back the answer, hence leading to $2x * (t_{co} + t_{ci}) + t_p * (x - 1)$. As a result, the reconstruction time required when polling is used is given by $t_{co} * (2x + 3) + t_{ci} * (l + 2x + 1) + t_p * (x - 1) + l * t_m$.

$$T_{notify\_ce\_id} = t_{co} \tag{4.8}$$

$$T_{reconstruct\_polling} = 2x * (t_{co} + t_{ci}) + t_p * (x - 1) \tag{4.9}$$

$$T_{delivery\_polling} = t_{co} * (2x + 3) + t_{ci} * (l + 2x + 1) + t_p * (x - 1) + l * t_m \tag{4.10}$$

By applying the same reasoning to compute $T_{delivery\_pushing}$, we find that t_{co} is required for $T_{notify\_subscriber}$ in order to send back to the subscriber the quadruple from the CE that satisfies the last SS. Indexing the ephemeral subscription is a request sent asynchronously without future. For this reason $T_{index\_ephemeral\_subscription}$ consumes t_{ci}. Then, to compute $T_{handle\_ephemeral\_subscription}$ we assume the worst case. Since the matching has been detected and an ephemeral subscription indexed, at least $q - l$ quadruples have been received by peers but only one is received by the subscriber. In other words, at most $q - 1$ matchings with an ephemeral subscription are still required to reconstruct the full CE and the same number of t_{co} to send back the chunks. Consequently, $(q - l) * (t_m + t_{co})$ is required for $T_{handle\_ephemeral\_subscription}$. Once times are added we get $t_{co} * (q - l) + q * t_m + l * t_{ci}$ for $T_{deliver\_pushing}$.

$$T_{notify\_subscriber} = t_{co} \tag{4.11}$$

$$T_{index\_ephemeral\_subscription} = t_{ci} \tag{4.12}$$

$$T_{handle\_ephemeral\_subscription} = (q-l) * (t_m + t_{co}) \tag{4.13}$$

$$T_{delivery\_pushing} = t_{co} * (q-l) + q * t_m + l * t_{ci} \tag{4.14}$$

Now, we write an inequation between $T_{delivery\_pushing}$ and $T_{delivery\_polling}$ to see under which value for x pushing is performing better than polling. Below is the result we get with the parameters introduced previously.

$$T_{delivery\_pushing} < T_{delivery\_polling} \tag{4.15}$$

$$\begin{aligned} t_{co} * (q-l) + q * t_m + l * t_{ci} < t_{co} * (2x+3) + t_{ci} * (l+2x+1) \\ + t_p * (x-1) + l * t_m \end{aligned} \tag{4.16}$$

$$x * (-2t_{ci} - 2t_{co} - t_p) < t_{co} * (l-q+3) + t_m * (l-q) - t_p + t_{ci} \tag{4.17}$$

$$x > \frac{t_{co} * (l-q+3) + t_m * (l-q) - t_p + t_{ci}}{-2t_{ci} - 2t_{co} - t_p} \tag{4.18}$$

Figure 4.3 depicts the possible values for x when we set parameters to $m = 1$, $t_{ci} = 12$, $t_{co} = 80$, $t_p = 500$ according to experiments and vary the number of quadruples per CE along with the subscription size. In that case we get $f(l,q) = (81*l - 81*q - 248)/-684$ and $T_{delivery\_pushing} < T_{delivery\_polling}$ when $x > f(l,q)$. In conclusion, the figure allows to deduce that the difference of performance strongly depends of the CE size whereas the subscription length has much less effect since the plan sketched by $f(l,q)$ is inclined and its slope mainly depends of parameter q. Also, the larger CE size is, the more the number of poll operations is required to have pushing that beats polling. This suggesting that pushing is beneficial for not so large CE sizes.

Unsubscribing Regarding a subscription S composed of l sub-subscriptions, to perform an unsubscribe operation consists in removing the indexed subscription S, the subscriptions originating from S (due to rewrite operations), the intermediate

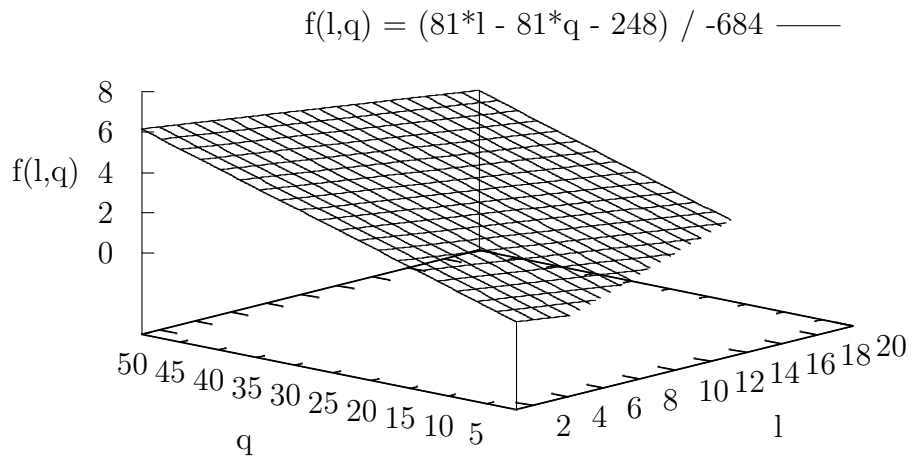


Figure 4.3 – Theoretical comparison between polling and pushing ($T_{delivery\_pushing} < T_{delivery\_polling}$ when $x > f(l, q)$).

results if the original subscription requires to receive notifications as *Bindings*, and finally to invalidate some caches used to improve access time to subscriptions.

To reach all the peers that contain the information to remove, two solutions are possible. Either an unsubscribe request is routed by following the path used to index S and each rewritten subscription S', S'', ..., $S^{(l-1)}$ recursively (each subscription has an identifier and rewritten ones know the identifier of their originator) or, it is possible to send an unsubscribe request to each sub-subscription directly. Indeed, each rewritten subscription is subsumed by a sub-subscription from S.

The former solution is interesting because a rewritten subscription S'' is supposed to contain less variables than S', hence the longer the chain is the less the number of peers to contact is. However, it implies to route a lot of requests. Supposing that n compound events have matched S, thus $n \times (l - 1)$ rewritten subscriptions are indexed and the same number of unsubscribe requests must be forwarded. In contrast, the latter solution needs only $l - 1$ requests to reach the necessary peers but because it uses the sub-subscriptions from S which have not been rewritten the set of peers to multicast is greater.

Finally, an unsubscribe operation is not atomic due to the multiple peers to

reach for updating their state. While an unsubscribe request is handled, the state of the subscribe proxy must be updated and a simple test added in the method called for receiving the notifications. The purpose of this update is to discard the notifications which are received for a subscription whose an unsubscribe request has been initiated.

OSMA

Regardless of the variant based on the first algorithm (polling or pushing) when a CE is published, the matching is not performed in parallel. Rather, it is initiated by the peer which stores the first SS. Let's say a subscription contains more than one SS, the matching with the second SS is not performed while the first SS is not satisfied, thus incurring a sequential evaluation. To alleviate this issue, we propose a second algorithm, called OSMA, which allows for parallel matching of same sub-subscriptions while avoiding the chain like approach.

The basic idea behind this second algorithm is to reduce the time spent to match a subscription by removing when possible the chain we had in the first algorithm and thus reducing the number of messages that are exchanged between peers. This version is optimized for the general, and hopefully most common case, where operations from a same proxy are received by peers in the same order they have been published. However, the new mechanism introduced with this algorithm can handle the temporal ordering issue, at a cost. In this new scheme, subscriptions and publications are handled as described in Listing 4.9. When a new CE is published, it is indexed by sending its whole content to each of the peers managing each one of the quadruples it contains. On the contrary, the subscriptions are still indexed as with the previous variants of the algorithm by forwarding a subscription to all the peers managing only **one** of the sub-subscriptions and for example, the first one. This behavior ensures that the matching between subscriptions and publications is performed locally without additional steps in the optimistic case where a subscription sent from a proxy before a publication is indexed before the publication. However, the local matching is performed at the extra cost of some more data to convey on the network. To choose only one sub-subscription for indexing the subscription is a sufficient condition because at least one quadruple

from a CE published through a proxy will reach the peer that contains the sub-subscription, if the CE is supposed to match the overall subscription. Upon the reception of a publication, a peer stores only the quadruple which has been used for routing the full CE, not all the quadruples. This ensures that at the end the quadruples from a same CE are eventually distributed on several peers as in the first algorithm. However, the full CE is also carried by the publication request to improve the time to detect the subscriptions which are satisfied. Therefore, just after the storage of the quadruple which has been received, we try to detect the subscriptions which are matched by using the full CE. However, a notification is sent back if the publication we are manipulating is not part of a CE which has already been handled. This condition is essential to ensure that two quadruples from the same CE indexed on the same peer and matching a same subscription do not generate duplicate notifications. Then, for each subscription found, we trigger a *OneStepNotifyRequest* to send back to the subscriber the chunks it is interested in.

Avoiding duplicates Similarly to the first algorithm, duplicate notifications may be generated. To prevent duplicates and ensure that only one peer sends the notification with this second algorithm, we apply the following rule. A peer notifies a match if and only if it is responsible for the first of the matching events contained by the CE. Let consider for the sake of the explanations a 2-dimensional CAN network with two peers. In this context, quadruples and SSs are pairs made of two values. Duplicates are possible if we have for instance $S = (?x, r)$ and a compound event $CE = (q_1, q_2, q_3)$ so that $q_1 = (d, d)$, $q_2 = (s, r)$ and $q_3 = (g, r)$. As depicted by Figure **??**, in that case the subscription is indexed on both peers. Since q_2 and q_3 are satisfying S, if the rule to avoid duplicates is not applied, each peer notifies independently the subscriber with the full CE. When the responsibility is checked, duplicates are prevented because q_2 which is indexed on the right side peer is the first quadruple in the CE list that matches S which is not the case for q_3. Consequently, the left side peer does not trigger a notification which avoids duplicates.

It is worth notice that the chain approach from the first algorithm is still used during the indexation of a subscription when it is detected as delayed regarding

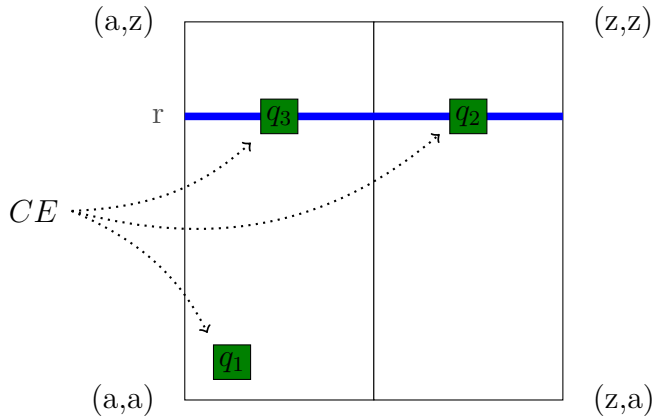


Figure 4.4 – Subscription (blue line) and CE (green boxes) mapping leading to duplicates.

some quadruples. This means that the subscriptions that are detected as matching a CE during the indexation of a publication may be a non-rewritten subscription or a rewritten subscription. The later case occurs when a subscription S published before a CE matching S is indexed between for example two quadruples of the same CE ($q_1 \rightarrow S \rightarrow q_2$). When S is indexed on the peer that contains q_1, q_1 is detected as matching S and S as delayed. Thus, S is rewritten by applying the same algorithm than the other variants. A full (one step) matching cannot be performed because we don't have the entire compound event. However, when q_2 is received with a *PublishQuadrupleRequest*, because we have at our disposal the full CE, we are able to short-circuit the chain algorithm to send back to the subscriber the right information. This is done implicitly by sending a *OneStepNotifyRequest* on line 35-36, regardless the subscription type (rewritten or not). Finally, if a subscription S is delayed and indexed after all the quadruples of a same CE matching S, then the chain algorithm is recursively applied on the rewritten subscriptions and as for the algorithm variant based on pushing, some ephemeral subscriptions have to be handled for CE listener.

```
1  # upon reception of a publication on a peer
2  def publish(compound_event):
3    indexation_time = now()
4    # set timestamp on each quadruple contained by the CE
```

```
5     compound_event.set_indexation_time(indexation_time)
6
7     for quadruple in compound_event:
8       # request carrying the full compound event to each peer
9       # managing a quadruple from the compound event
10      async_send(
11          IndexPublishQuadrupleRequest(compound_event, quadruple))
12
13    # the meta-quadruple is required in case we fallback to CSMA
14    # due to ordering issues
15    meta_quadruple =
16        create_meta_quad(compound_event, indexation_time)
17    async_send(PublishQuadrupleRequest(meta_quadruple))
18
19  # upon reception of an index publish quadruple request
20  def receive(publish_request):
21    compound_event = publish_request.compound_event
22    quadruple = publish_request.quadruple
23
24    store(quadruple)
25
26    subscriptions = find_subscriptions_matching(compound_event)
27
28    for subscription in subscriptions:
29      # create notification content according to the
30      # subscription type and variables
31      chunks = filter(compound_event, subscription)
32      # notifies a match if and only if the peer is responsible
33      # for the first quadruple matching the first sub-subscription
34      if responsible(subscription, compound_event):
35        async_send(subscription.proxy_url,
36            OneStepNotifyRequest(subscription.id, chunks)
37
38      # handle ephemeral subscriptions that are waiting for
39      # a result due to the ordering issue
40    ephemeral_subscriptions =
41      find_ephemeral_subscriptions_matching(quadruple)
42    for ephemeral_subscription in ephemeral_subscriptions:
43      async_send(ephemeral_subscription.proxy_url,
```

```
44          NotifyRequest(ephemeral_subscription.id, quadruple))
45
46  # upon reception of a notification on a subscribe proxy
47  def receive(notification):
48    subscription = find_subscription(notification.subscription_id)
49    listener = find_listener(notification.subscription_id)
50    listener_type = type(listener)
51
52    # the notification which is received contains all the necessary
53    # chunks due to a matching performed in one step
54    if type(notification) is OneStepNotifyRequest:
55      # create the result to deliver according to
56      # the type of the notification listener
57      result = create_result(notification, listener.type)
58      listener.deliver(subscription.id, result)
59      async_send(
60          RemoveEphemeralSubscriptionRequest(
61              notification.quadruples[0].graph))
62    else:
63      # same as Listing 4.8 from line 55 to 91
```

Listing 4.9 – Publishing and subscribing with OSMA. Only methods which have been edited compared to CSMA are showed. Parts that differ are highlighted.

The main benefit of this second algorithm is the expected low latency for subscribers. As soon as the CE reaches the peer responsible for the first matching event, a notification is triggered. Also there is no need for a reconstruction phase because the Compound Event can be directly sent to the subscriber. However, this is done at the cost of bandwidth since the whole Compound Event is sent to multiple peers. Besides, note that this second algorithm cannot deal with the situation where a subscription is created before an event but reaches a peer after. Correctly managing this case requires falling back to CSMA, which we do.

4.3 Evaluation

The experiments introduced hereafter have been performed on 29 nodes of the Grid'5000 testbed. Each machine embeds a Xeon E5520 @ 2,26 GHz with 32

GB RAM, a hard disk drive at 7200 RPM. The partition used for data storage is an EXT3 partition mounted with options *noatime* and *nobarrier* for performance reasons. Java 7 was used with JVM option *-server*. Each result is the average execution on 6 runs where the first run is laid aside due to JVM warmup.

The workload we have used is made of x synthetic events and y subscriptions that are generated to be distributed uniformly among the available peers. This allows us to evaluate the performance of the algorithms when the number of peers involved is the largest. Subscriptions are generated to embed k quadruple patterns of the form $(?g, ?s1, p1, ?o1) \wedge (?g, ?o1, p2, ?o2) \wedge \ldots \wedge (?g, ?o_{k-1}, p_k, ?o_k)$. Compound Events are generated to evenly match subscriptions by affecting approximatively the same number of quadruples per peer. Although quadruples are synthetics, the distribution is not perfect due to the graph value that is shared among quadruples from a same CE but also because CEs are generated to match path queries.

Before entering into the explanation of the different experiments we made, it is worthwhile to explain the different values we have measured and which ones we retain to compare algorithms. Figure 4.5 summarizes a simple benchmark configuration where one publisher publishes N compound events and a subscriber subscribes to consume all the events published. Publications are indicated as $p_1, ..., p_N$, notifications as $n_1, ..., n_N$ while $T_p(p_i)$ and $T_d(n_i)$ represent respectively the time at which the publication i has been published from the publisher and the time at which notification n_i has been delivered on the subscriber. With this simple configuration we have defined several metrics to compare the performance of the algorithms:

End-to-End throughput gives the number of events handled per unit of time from an end-to-end perspective. Mathematically, it is defined by computing the formula $N/(T_d(n_N) - T_p(p_1))$ where $T_d(n_N) - T_p(p_1)$ is the time elapsed between the first publication from the publisher $T_p(p_1)$ and the last notification received on the subscriber $T_d(n_N)$.

Subscriber throughput gives the number of events handled per unit of time from a subscriber point of view. It is computed as $N/(T_d(n_N) - T_d(n_1))$ where $T_d(n_N) - T_d(n_1)$ is the time elapsed between the first notification received

by the subscriber $T_d(n_1)$ and the last notification received by the subscriber $T_d(n_N)$.

Point-to-Point latency gives an idea of the average latency required for an event to be published, handled by the brokering system and delivered to the subscriber. It is computed with the formula $\left(\sum_{i=1}^{N} T_d(n_i) - T_p(p_i)\right)/N$.

Each measurement is valuable and shows a critical characteristic of the system. Although all measurements are computed, we consider mostly the subscriber throughput in the following experimentations in order to keep the explanations concise.

In the first experiment we evaluate the effect of increasing the network size. For this purpose we place one peer per machine and vary the total number of peers from 1 to 25. There is only one subscriber with a subscription made of $k = 5$ patterns. One publisher publishes 3×10^3 CEs, each one containing 5 quadruples for an approximate size of 670 Bytes. Figure 4.6(a) depicts the average subscriber throughput, i.e. the throughput perceived on the subscriber when the network size is increased. OSMA outperforms CSMA by a factor of 5.43 according to the median value. This difference is explained by the matching which is performed in one step with OSMA whereas CSMA requires a number of steps equals to the number of SS contained by a subscription that is satisfied. Thus, increasing the number of routing steps required.

In a second experiment we evaluate the effect of varying the number of publications. Figure 4.6(b) shows that the throughput on the subscriber is constantly increasing with OSMA when the number of publications increases. This is because the overlay is not working at its full capacity when $x = 30 \times 10^3$ CEs are published. On the contrary with CSMA the subscriber throughput decreases quickly with the number of publications. This behavior is explained by the reconstruction process which overloads peers with requests, slowing the publications and the notifications. Because the time required to complete the experiments is too large when more than 21000 CEs are published with CSMA, some values are omitted.

The third experiment evaluates the impact of varying the number of subscriptions registered in the system. The scenario consists in one subscriber subscribing

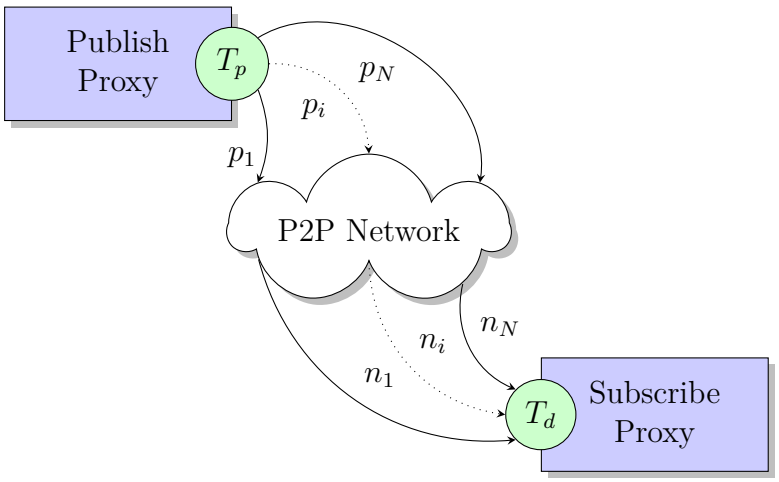
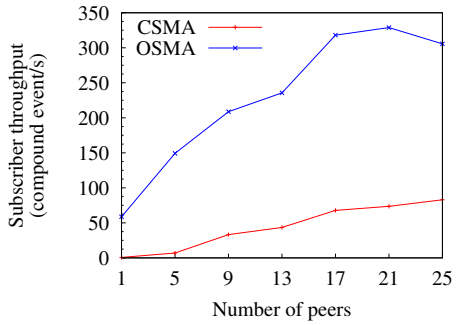


Figure 4.5 – Possible measurements to evaluate and compare the proposed publish/subscribe algorithms.

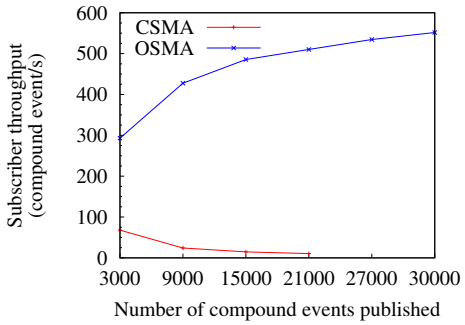
with various number of subscriptions. The subscriptions are generated to match when possible an equal number of Compound Events.

Figure 4.6(c) shows the subscriber throughput for 1 to 60 subscriptions. With OSMA the throughput decreases almost linearly with the number of subscriptions in the system. The reason lies in the indexing of the subscription. Since it relies on the first sub-subscription which contains only a predicate as fixed term, only half of the peers of the overlay are actually participating in the matching. Also, some of them have multiple subscriptions to check for each Compound Event received, which is a costly operation with the underlying storage engine we are using. On the contrary, CSMA remains almost stable with a throughput that varies around 92 CEs per second. This effect may be explained by the rewritten subscriptions that are generated once a first sub-subscription is satisfied. A rewritten subscription contains in our case more fixed parts than its parent and is indexed against potentially less and on different peers, thus, increasing the number of peers involved in the matching.

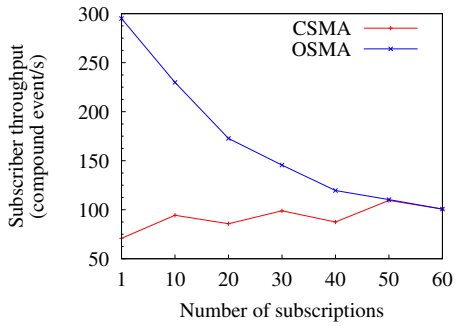
In a fourth experiment we test the effect of varying the number of peers when selective subscriptions are replaced by a subscription that accepts all events (cf. Figure 4.6(d)). In such a situation, all peers index the subscription represented by the SPARQL query `SELECT ?g WHERE { GRAPH ?g { ?s ?p ?o }}`. As ex-



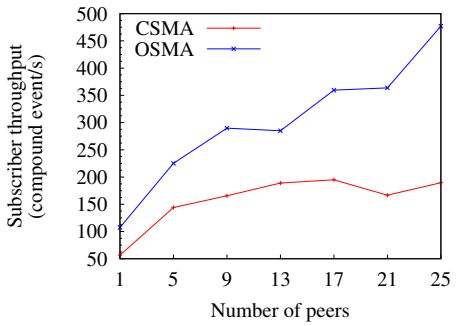
(a) Impact of overlay size. 3000 CEs published, one subscription of $k = 5$ quadruple patterns.



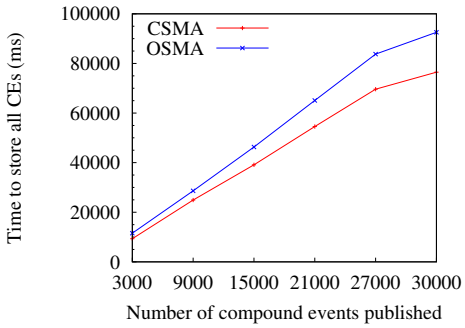
(b) Impact of the number of publications. 25 peers, one subscription ($k = 5$).



(c) Impact of the number of subscriptions. 25 peers and 3000 CEs published.



(d) Scalability with one accept-all subscription. 3000 CEs published.



(e) Time to store publications on peers. 25 peers, no subscription, 25 quadruples per CE.

Figure 4.6 – Performance comparison of CSMA and OSMA.

plained in Section 4.2.3, CSMA generates a lot of duplicate notifications in this situation, which limits the scalability. Since OSMA always performs a single notification, the throughput increases with the number of peers.

In a penultimate experiment, we measure the time taken to store different number of publications with no subscription registered on peers. Results are depicted on Figure 4.6(e). Unlike previous experiments CE size is increased to 25 quadruples. The time required to complete benchmark runs is directly related to the bandwidth consumption since no matching is performed. Indeed, the only difference between the two algorithms with this configuration is the quantity of information conveyed from peers to peers. The time to store the published events quickly differs between CSMA and OSMA when the number of publications increases from 3000 to 30000. It confirms that OSMA requires more time than CSMA to forward events to peers that are responsible to store quadruples. Thus, it will require more bandwidth than CSMA.

Finally, we compare *CompoundEvent*, *Binding* and *Signal* listeners on a eight peers configuration. When OSMA is used, we get that subscribing with a *Binding* or *Signal* listener never increases performances by more than 3% compared to the case where a *CompoundEvent* listener is used. The reason lies in the fact that OSMA applies almost the same scheme whatever the listener used is. The clear difference is the size of the answer sent back from peers to subscribers. Since our configuration uses gigabit Ethernet, it explains the small effect. Scenarios with limited bandwith between subscribers and peers will most probably benefit from choosing the right listener according to their needs. The impact is much different when CSMA is used because the actions but also the number of communications required to notify solutions greatly differ. Our tests show that using a *Signal* or *Binding* listener increases the subscriber throughput by at least 5, thus strengthening the reason to have different subscription listeners.

In conclusion, the experiments show that OSMA outperforms CSMA in terms of throughput and scalability at the cost of a higher bandwidth consumption. Its only limitation is that it cannot enforce the happen-before relation and hence, depending on the use case, some applications will have to rely on CSMA.

Summary

In this chapter we have introduced a publish/subscribe infrastructure based on the RDF data model and SPARQL filter model. Subscribers can express their interests using a subset of the SPARQL language and events are published as RDF data. We rely on a multi-dimensional indexing space and lexicographical order to distribute both the publications and subscriptions on an overlay. Compared to previous works, our scheme does not require multiple indexing of the same publication, thus reducing the storage space. We have proposed two algorithms for matching subscriptions with events. The first one, CSMA, is based on the canonical chain like approach. It reduces the bandwidth used when publishing at the cost of a longer matching time. It can also handle ordering issues which can happen when a same client submits both publications and subscriptions. The second one, OSMA, uses a fully distributed approach which leads to good performance at the cost of a slightly heavier publication process. To summarize, the different properties of the two algorithms are presented in Table 4.1. Both algorithms have been experimentally tested for throughput and scalability.

	Routed Element	**Matching Steps**	**Duplicates**	**Happen-Before**
CSMA	Individual quadruples	Multiple, Chain-like and Reconstruction	Yes, filtering required	Enforced
OSMA	Whole *Compound Event*	Single	No	Requires CSMA

Table 4.1 – Comparison of the two publish/subscribe algorithms proposed.

Chapter 5

Distributed RDF Load Balancing

Contents

In the previous chapters we have introduced a solution for respectively storing, querying RDF data and selectively disseminating RDF events. In this context, RDF information used for the experiments was synthetic because our desire was to assess the throughput of our solutions when the system works at its maximum capacity. However, real RDF data is highly skewed and it affects how information may be shared between nodes. The issue is caused by some RDF terms that are more frequent than others, especially predicates. Usually, hashing is employed to enhance distribution however this last is pointless when the imbalance is caused by popular terms since the same inputs produce the same hash values. This chapter introduces a solution to distribute fairly, with our architecture, RDF data which experiences a high degree of skewness. First, we introduce some related works

about load balancing in structured P2P networks to describe common strategies. Then, we position ourselves and we describe our solution by identifying what are our criteria and the different mechanisms involved. Finally, we conclude with some experiments which assess the utility of our solution with real workloads.

5.1 Related Works

Load balancing is at the heart of many P2P works to address the performance issues that stem from load imbalance on peers. Imbalances may be caused by an unfair partitioning of network identifiers between peers, frequent arrival and departure of peers but also the heterogeneity in terms of bandwidth, storage and processing capacity between machines where peers are deployed. Other reasons can be related to the variation of size, popularity and lexicographic similarities among resources handled by P2P networks (the last three being critical with RDF data). The works that consider this area of research can be classified into two main groups; either static or dynamic. In the former, the system load is assumed stable. No continuous insertions or deletions are performed and queries remain similar. Solutions are sometimes based strictly on a fixed and preconfigured set of rules. Furthermore, churn is often evicted and the load balancing decision is assumed to be taken during the join of a peer. The latter enables decisions and adaptations at runtime while taking into account endless data insertions but also turnover among peers, namely arrival and departure.

In structured P2P networks, peers manage part of a common identifier space, which is a circle segment, an hypercube subset or a subtree. Usually, resources or data that have to be indexed into the network are assigned to an identifier from the common identifier space to enable routing based on the range a peer is responsible for. This identifier can be the data itself or a hash value associated to the information when a DHT or consistent hashing are at the basis of the indexing scheme. Eventually, the information is indexed on the peer managing the resource identifier. To address load balancing issues in structured P2P networks, especially regarding the distribution of data, several load balancing strategies have been proposed based on replication or relocation. The model followed by the strategies usually consists in controlling resources location, peers location or both.

However, many variants are conceivable based on indirections, identifiers or range space reassignation and virtual peers[1]. Besides, designing a load balancing solution requires to consider additional parameters such as the overload criteria to take into account, how overload is detected and how load information is exchanged. This variety of parameters has led to the definition of multiple solutions that sometimes differ by minor but subtle changes. Hereafter are introduced some solutions we find relevant in our context.

5.1.1 Static load balancing

Rao et al.

In [136] the authors suggest three different strategies based on virtual peers to address the issue of load balancing in P2P systems that provide a DHT abstraction. As the load balancing issue is hard to address in its full generality they make the assumption that the load imbalance is due to the lack of one resource only: storage, bandwidth or CPU. Moreover, their solutions are supposed to transfer the load between highly loaded and lightly nodes by moving virtual peers only and the load on virtual peers is assumed stable during the execution of their load balancing scheme. Furthermore, the system is considered static in the sense that peers do not join and leave the system continuously.

Their first scheme called one-to-one involves two peers to decide whether a load transfer must be performed or not. The process is basic and simply consists of contacting a randonly chosen peer. If the peer that received the message is heavily loaded, then a transfer may take place between the two nodes. The second scheme relies on directories indexed on top of the existing overlay. These directories form a meta overlay where each directory, indexed on a node, is in charge of maintaining the load that may be reported. Load information about virtual peers is piggybacked by periodic advertisements sent from lightly loaded nodes. The assignment between light nodes and directories do not change over time, as the

[1]Virtual peers is an abstraction for several peers hosted on a same physical node/machine. Traditionally in a P2P network, a peer is a node or machine. On the contrary, a virtual peer is a peer that can be deployed with some other virtual peers on the 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.

number of directories is static. Periodically, the heavy nodes also report virtual peers load to their specific assigned directory. This request is also used as a sampling request to examine the directory where the load is reported. Indeed, once a directory receives a sampling request from a heavily loaded node, it looks at the local entries reported by lightly loaded nodes to find the best virtual peer that can be transferred from the heavily loaded node to a lightly loaded node contained by the directory. Thus, in contrast to the first scheme, this one-to-many approach considers several lightly loaded nodes to make the transfer decision. Their third variant extends the first two by matching many heavily loaded nodes to many lightly loaded nodes. Similarly to the second scheme, directories are used. Each node reports periodically its complete load information to a given directory. Once a directory has received enough information from nodes, it triggers a three phases algorithm that consists of *a*) transferring virtual peers from heavily loaded nodes to a global pool; *b*) reassigning virtual peers from the pool to lightly loaded nodes without creating any new heavily loaded nodes; *c*) dislodging virtual peers that have not been reassigned during phase *b* by swapping the largest virtual peer (in terms of load) from the pool with a lightly loaded node and coming back to step *b* while some entries remain in the global pool.

Their simulation results show that the first two approaches are able to balance the load within 80% of the optimal value and the third based on many-to-many sampling can achieve 95% of the optimal value. However, these results are achieved by performing several load balancing rounds, from around 50 with the best scheme to 20000 with the worst scheme depicted by the one-to-one strategy, but no indication is given regarding the convergence in terms of execution time.

Bayers et al.

In [137] the authors investigate the direct applicability of the power of two choices paradigm [138] on the Chord P2P network for addressing load imbalances in terms of items per peer. The authors debate over the approach taken in the original Chord paper that consists of using virtual peers. They make an analogy with the standard balls and bins problem [139] with n items and n peers to show that even with perfectly uniform assignments of the circle segments to peers, the load

remains not well balanced. In addition, they claim that using at least $O(\log n)$ virtual peers per node leads to a high number of neighbors to maintain per peer which is not acceptable due to heartbeat messages that are exchanged periodically to detect failures.

The scheme they applied to balance the load between peers can be summarized in a few lines. The node that wishes to insert an item applies d hash functions on the item key and gets back d identifiers (each hash function is assumed to map items onto a ring identifier). Afterwards, a probing request is sent in parallel for each identifier from the identifiers computed previously and the peers managing the identifiers answer with their load. Once load information is retrieved, the peer with the lowest load is adopted for indexing the item. The lookup operation from an item key is similar to the insertion and consists of querying the d peers whose at most one will successfully locate the item. While the search operation is parallelizable, the authors care about the network traffic provoked by get operations and propose a simple variant. In addition to storing the item at the least loaded peer p_i, this variant consists of adding a redirection pointer to p_i on all other peers p_j where $j \neq i$. Thus, a lookup can be achieved by using only one hash function among d at random. 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 and provides almost the same benefits as using an unlimited number of virtual peers. However, their context is restricted to items with equal size and popularity. Moreover, the proof they give about the maximum load expected on a peer with high probability, when the two choices method approach is used, rests upon the previous mentioned restriction and the fact that items are inserted sequentially. In the paper it is not very clear whether this restriction holds for all the introduced properties or not. Last but not least, to ensure recovery from crashes but also to prevent expensive mechanisms for keeping references up to date in case of churn, the variant of their insertion scheme that uses redirection pointers assumes a soft state approach (i.e. the keys and their value are periodically re-inserted) which limits the applicability of their solution.

Meghdoot

Closer to our work in terms of network topology considered, the authors exploit in [122] the characteristics of CAN and their publish/subscribe system (Meghdoot) properties to balance the load when new peers are admitted into the system. They distinguish subscriptions load from events load given that they have to be handled differently. In the former case, subscriptions load on a peer is proportional to the number of subscriptions stored on it. Reducing the number of subscriptions on a peer decreases the load on a peer. Thus, their idea is to split the peer zone so that the number of subscriptions is evenly divided with the peer that joins. The latter case addresses the load imbalance with events. The new and the old zone may fall in the propagation path of many events and splitting the zone as for subscriptions may, in addition to the existing peer, overload the peer that joins. Therefore, the authors propose to create 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, instead of shared to p_j along with 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 replica peer out of the list of replicas in a round robin fashion. This replication strategy improves load balancing, data availability and performances. Also, to locate a heavily loaded node during a join operation, each peer in the system propagates periodically its load with its neighbors. This knowledge is then used during the join procedure to forward the operation to the heaviest loaded neighbor step by step until to reach a peer that has a load higher than any of its neighbors.

Similarly to other static solutions, the authors assume the existence of an oracle to decide when peers have to join the system. However, the system is not able to adapt itself to improve the load imbalance. As they explain for their experiments, they define an injection period where a new peer joins the system after each simulated event with a probability of 10%, until the total number of peers reaches an expected bound.

RCAN

In [140] the authors introduce a solution for improving load imbalance on an extended version of CAN. Their system named RCAN [141] differs with CAN regarding the neighbor entries maintained on each peer for routing the requests. In addition to the standard immediate neighbors, every peer controls a number of links towards peers in the system that are at distance inverse of the power of 2 along each dimension of the identifier space. The number of links maintained per dimension is assumed to be in $O(\log n)$ where n is the number of zones in the system, when the multi-dimensional space is split uniformly. These additional neighbors have the benefit to improve the routing complexity because instead of applying a greedy scheme where a request is forwarded to an adjacent neighbor step by step, the request bypasses some stages similarly to Chord with its finger table. Regarding load balancing, the authors leverage the additional long links that each peer maintains to probe faraway peers periodically without having to send at random messages that are routed through several intermediate peers. Then, these samples are reused during a join operation to know what is the best peer to join for improving the load imbalance. In this paper the authors make, as for previous solutions, strong assumptions about the machines that are assumed to be homogeneous, the data that is supposed to have the same size and popularity but also the load balancing decision that is taken during join operations.

Battre et al.

A few attempts have been made about load balancing with RDF data. Battre *et al.* propose in [92] a solution for solving the bad distribution of popular RDF terms on DHT. It consists of creating an overlay tree atop the existing Pastry [22] overlay at the cost of more indirections and datastores to maintain per peer. Peers are assumed capable of detecting overloads. Upon the detection of a load imbalance, a peer splits its current dataset into two parts. The first half remains on the current peer database along with a reference to a new peer that contains another database to store the second half of data that is transferred. Due to the new references that are attached to peers, further steps are required to resolve queries. Thus, the evaluation of a query consists of following the new references and looking at the

new datastores in addition to the standard lookup mechanism. This leads to a query resolution in $O(\log n + d)$ where n denotes the number of nodes in the DHT and d the depth of the tree. No experimental result nor evaluation methodology is given.

RDFPeers

In RDFPeers [90], the authors decide to simply ignore popular RDF terms and do not use them for indexing data. The authors make an analogy with English language where the words "a" and "the" occur frequently but are not valuable as search terms. However, we think this comparison is not appropriate. For instance, RDF predicate *rdf:type* is a popular resource for reasoning purposes, and to evade this term implies to preclude queries or subscriptions that perform filtering on this term. Although the authors rely on hashing to leverage the uniform keys distribution among peers, the frequency count distribution of non-popular RDF terms is still skewed and the difference between the minimum and maximum number of data contained by each peer remains large. To achieve a better load imbalance they propose a load balancing scheme based on successors probing. Their solution derives from [142] and aims to provide peer keys distribution adaptive to the data distribution. It consists essentially in performing random walks by creating a set of keys that are used to route probing requests. Upon the reception of a probing request, the peer returns the numbers of RDF resources that would be transferred if a join was performed (by assuming the interval managed by the peer that is joined is split into two). Once the results are gathered on the request initiator, the new node joins the system by using the key that transfers the heaviest load in terms of triples. The experiments the authors provide show their scheme reduces load imbalance to much less than an order of magnitude.

5.1.2 Dynamic load balancing

A second wave of works based on dynamic load balancing has been proposed. Most of them are theoretical and try to provide a guarantee on the maximum imbalance and load moved on the system at any time.

Godfrey et al.

Godfrey *et al.* propose in [143] an extension of a previous work on static load balancing [136] (introduced in Section 5.1.1). This work complements the last by considering dynamic structured P2P systems. In this way they relax some assumptions to allow continuous data insertion and deletion, peers churn and the skewed distribution of data during load balancing decision. Their purpose remains the same and consists of minimizing the load imbalance and the amount of load moved. As in [136] they rely on virtual peers to move load and they assume one bottleneck only in the system. Besides, they leverage their previous many-to-many scheme (for periodic load balancing) combined with an additional emergency threshold to boost the decision mechanism.

Bienkowsky et al.

In [144] the authors focus on structured P2P networks based on a ring topology. They explain that the communication load and the amount of data a peer stores depends heavily on the length of the interval it manages. That way they define a smoothness parameter that depicts the average interval length managed by peers in the system with the goal to have each peer managing an interval whose the length is the closest to the smoothness parameter. To accomplish their aim they count on a randomized algorithm where each peer probes periodically during a predefined number of rounds another peer managing an identifier selected at random. Then, according to the probing information and predecessor information they can round by round force peers to join and leave the network to reach their ideal smoothness value. They prove their distributed scheme works with high probability and that its cost in terms of peer migration is optimal. However, their algorithm implies to estimate the total number of peers in the system, to sometimes block some peers for a few rounds and to tune several parameters. Furthermore, in contrary to the previous work, node churn is not allowed during rebalancing.

Vu et al.

More recently, Vu *et al.* propose in [145] a structured P2P agnostic solution for load balancing in dynamic context based on histograms. The peers are bundled

into non-overlapping groups that constitute the bins of the histogram maintained on each peer. The histogram acts as an approximate global view of the system to know the load distribution. The load propagation follows a gossip scheme where a peer forwards its updated load if the ratio between the new load and the load sent before is greater than a parameter M. They prove that if the maximum load imbalance between a peer and the average load of a group of peers from a histogram is k, then the maximum load imbalance ratio of the system is k^2. In addition, they give a relation between M and k so that M can be chosen to keep the maximum load imbalance ratio of the system under a given threshold. A peer is detected as overloaded or underloaded when its load is respectively twice the average load of any group in its histogram or half the average load of any group in its histogram. Although their system makes explicit some interesting properties, it remains sometimes unclear how non-overlapping groups are created and maintained dynamically.

Mercury

In [24] the authors present Mercury, a system made to support range queries on top of a structured P2P network constructed by using multiple interconnected ring layers where each one is named a hub. Each hub manages the indexation of an attribute from a predefined schema. 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 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 request has a TTL value set to $\log n$ where n is an approximate value of the number of peers in the system used to know when the routing steps must stop. Furthermore, peers periodically probe their d-neighborhood. The combination of these methods offers a global system load assessment whose values are collected into histograms maintained on peers. 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.

5.2 Load Balancing Solution

In this section we introduce and discuss the different mechanisms considered and applied for addressing RDF load balancing issues that occur with the architecture we propose. These mechanisms focus in a first time on improving the bad distribution of RDF data with the purpose at the ende to enhance the involvement of peers with the publish/subscribe matching algorithms.

To better understand the options we have but also the choices we have made, it is worth explaining why some RDF terms are more popular than others. Let's start with the unequal popularity of predicates. This last may be explained by how semantic is added to RDF data through the definition of vocabularies (i.e. ontologies). Ontologies allow to describe different concepts but each concept may have many instances. In concrete terms, a processor is a concept characterized by many properties such as its model name, number of cores, etc. However, many instances exist, that is, different processors having different properties. Since properties are usually identified in the RDF model as predicates, multiple instances may reuse the same predicates. In a lower degree, popularity can also be caused by the same value shared between tuples' subjects and objects. Sharing RDF terms allows to link pieces of information modeled in RDF. Therefore, the bigger and the more frequently used an ontology is, the greater the probability to have skewed RDF terms becomes.

Previously, the emphasis was placed on RDF terms popularity. The reason lies in the fact that whatever approach is taken to spread RDF data on nodes, using hashing or not, the final distribution is bad. However, systems that do not rely on hashing, as the one we propose which indexes data based on the lexicographic order, are exposed to one additional problem. The issue is about IRI prefixes used with RDF terms that are often similar. Using hashing solves the problem because even small changes in input values generate different hash values that are distributed uniformly with high probability in the identifier space managed by peers. The situation is different when the lexicographic order is used. RDF

terms with same prefixes force data to be indexed on the same or successive peers. Balancing data thus requires in that case to change the identifier range managed by peers.

In the next section we identify and describe the stages through which we must pass in order to define our load balancing solution, along with the different options that are conceivable and the ones which have been selected. To limit the number of alternatives but also to position our solution regarding existing works, we have made some decisions based on our system's properties and the context where it is used. First, our main goal is to balance RDF data added synchronously or published asynchronously to the system. To this aim, our approach relies on peers relocation and not replication since this last is well known to improve data access by balancing queries load but not data itself. Second, we assume that no knowledge about ontologies associated to information received by the system is available because some sources are hidding this information. The argument is that vocabularies allow to infer confidential facts which is for instance forbidden when sharing medical data across Europe. Since no upstream knowledge is available, balancing is assumed to appear after data has reached its final destination, not at the entrance of the system. Third, to leverage the existing join and leave operations our solution is assumed to rely on virtual peers and therefore ousts solutions based on data relocation only which incur expensive mechanisms to maintain up to date pointers. One additional alternative to virtual peers would be to shift bounds managed by peers without moving them from a node to another with the help of a join or leave operation. This strategy is part of a thesis started by Maeva Antoine. More details will be available in her forthcoming manuscript. Fourth, peers are assumed deployed on homogeneous machines. It is a reasonable hypothesis since the middleware we propose targets deployment over a cluster. Finally, our solution aims to support dynamic load balancing, that is autonomous balancing decisions at runtime.

5.2.1 Options and choices

Designing a load balancing solution involves multiple stages and many potential alternatives we have identified and summarized in the following. The steps that

are described also explain some of our choices and act as the basis to comprehend the strategies we propose in the next section.

Detecting load imbalances

Before balancing load imbalances, disproportion in terms of load must be detected. It implies to know which load criteria are involved and how their variation could be measured on peers. Once these first questions are answered, the next step consists in deciding if a peer is enduring an heavy amount of work with respect to the selected criteria.

Measuring load Previously, we mentioned that our main goal is to adjust load for RDF data handled by the system. In this context our main criterion is the number of quadruples per peer. However, when working with RDF data it is clear that some quadruples are bigger than others. This is especially true for quadruples with literal object values that permit unbounded plain text descriptions. Even though quadruples' length depends of information sources, our system is storing on disk all incoming data. Since disks are resources with limited capacity, it could be worthy to consider quadruples' size as an additional criterion. To strengthen our thought, let's consider a case where a lot of quadruples are stored on a first peer and a few on a second. In that case, disk space consumption may be greater on the second than the first if quadruples on the second peers have larger RDF terms. Consequently, a second criteria is defined for disk space consumption. The load is measured for the two aforementioned criteria. The measurement for the first criterion is accomplished by recording the number of quadruples handled and stored by a peer. However, the second measurement is more subtle and must not be computed by summing RDF terms' size. The reason lies in the fact that most centralized RDF engines do not store quadruples as they come but rather by using indirection tables that eliminate duplicated RDF terms to prevent excessive disk usage. Therefore, disk consumption is measured by computing the ratio between the number of megabytes written on the disk by the peer local storage and the partition size in megabytes where the data is located.

Deciding about imbalance The next stage is about the process involved to detect whether a peer is experiencing an imbalance or not. In general, taking decision requires to compare loads between peers to deduce their state. Intuitively, the states in which a peer may fall are *overloaded*, *underloaded* or *normal*, i.e. not overloaded neither underloaded. However, not all load balancing solutions define and detect underloads. In our case we think it may be useful because RDF terms have different popularities and clusters of quadruples often emerge. Concretely, this is materialized by one or a few adjacent peers from the identifier space receiving all quadruples and many indexing no information. Figure 5.1 depicts a CAN cutting that exhibits the issue. Peers $2, 4, 5$ and 6 could be detected as *underloaded* regarding others. In that case peer 5 could leave and rejoin peer 7 to offload half of its load. Also, peers 2 and 1 could be merged in order to reduce routing steps but it is out of the scope according to the criteria specified above.

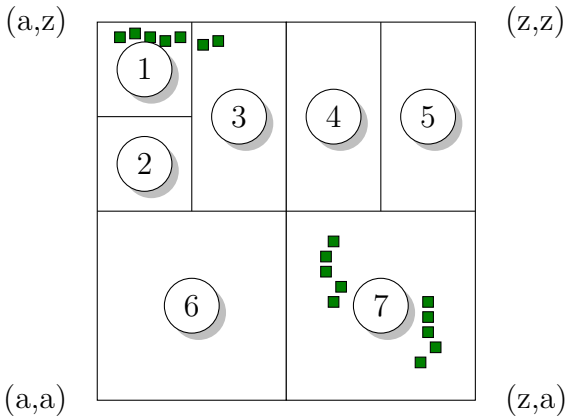


Figure 5.1 – RDF data clusters on a 2D CAN network.

The natural continuation is to explain how a peer knows that it is overloaded or underloaded. The process can be summed up by the function sketched on Algorithm 5.1. It takes five parameters:

- C, a criteria list sorted by descending order according to priority in which imbalances are handled. Criteria are assumed static and defined before the system starts;
- M, a load measurements list containing load measurements associated to

criteria defined in C. Measurements are assumed to represent load snapshots per criteria at the time the imbalance must be detected;

- E, a system load estimation list containing the system load estimation value for each criteria defined in C. This list plays a key role in the decision process. How it is built is discussed hereafter;

- K_1 and K_2, parameter constants defined per criteria before the system starts up. Their purpose is to keep the load per criteria close to a factor. The parameters also prevent oscillations. By setting them to high values, imbalances are spotted less often.

Based on parameters, the function respectively detects a peer as *overloaded* or *underloaded* if its load for a criteria $C[i]$ is respectively $K_1[i]$ times greater or $K_2[i]$ times lower than the estimate $E[i]$ associated to the criteria $C[i]$ that is observed ($K_2[i]$ must be lower or equals to $K_1[i]$). The order in which criteria are defined matters since it defines priorities in which imbalances are detected. 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 actions required to fix imbalances depends of criteria. To conclude, while remaining simple, this load detection model easily supports the definition of multiple independent criteria.

Regarding parameter E, its purpose is to offer an approximate value of the system load per criteria. By choosing E values close to the average system load, a peer may compare its load per criteria according to values in E and react accordingly to balance the load (by using for instance the schemes proposed in the next section) in order to keep the load per criteria close to a factor k (when $k = K_1[i] = K_2[i]$). Therefore, how E values are computed is critical. Many solutions are conceivable. The simple one is to use purely local knowledge by defining constant values for E. For instance, if a criteria is the number of quadruples per peer, the estimate value could be set to the number of peers divided by the total number of events the system is assumed to receive. Although this solution is merely static in the sense that some system's parameters are known before its instantiation (e.g. number of peers and total number of data to handle), it may be useful in the context where

the system is not so dynamic and the number of machines limited. In that case the system starts from one machine and new ones are added when the first reaches its maximum capacity defined by threshold values in E.

```
1: function EvaluateLoadState(C, M, E, K_1, K_2)
2:     ▷ i, load measurement index
3:     ▷ m, load measurement value
4:     for i, m ∈ M do
5:         if m ⩾ E[i] × K_1[i] then
6:             return LoadState(Overloaded, C[i])
7:         end if
8:         if m < E[i] × K_2[i] then
9:             return LoadState(Underloaded, C[i])
10:        end if
11:    end for
12:    return LoadState(Normal)
13: end function
```

Algorithm 5.1 – Load state estimation algorithm.

The second alternative is to populate E with values that represent the average system load per criteria. It implies to share knowledge about loads between peers. To disseminate these information messages must be exchanged and as a result a *pull* or *push* approach is possible. The push model was selected for two reasons. The first is that only an approximation is required, thus receiving new load values in time for computing an average result or later due to the asynchronism of the push model is not strictly speaking an issue. The second reason lies in the fact that probing peers with synchronous requests incurs higher bandwidth consumption because of roundtrip. Once the model to appraise peers utilization is defined, many solutions still exist to disseminate load information. Load may be piggybacked by usual requests but it implies that the convergence time about the rumor that relates to the load depends of the system usage which prevents decisions when part is idling. More simply the load can be periodically forwarded to immediate neighbors, the k-neighbors, broadcasted to all the peers or sent to peers selected at random or based on heuristics. Works made around gossips protocols are a great source of inspiration [146]. As we will see, the gossip protocol used is the main differentiation parameter for the strategies we propose in Section 5.2.2.

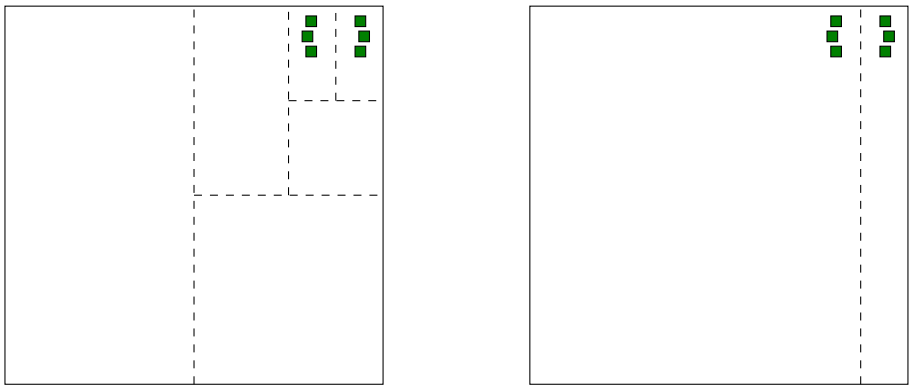
Balancing the load

Once an overload or underload is detected, the next stage is to fix it. To this aim, each criteria is associated to two functions. The first defines how to correct underloads whereas the second details how to regulate an overload. Whatever the implementation is, both methods require to select who will receive part of the imbalance and to define how the imbalance may be fairly shared. The next two paragraphs describe these two actions.

Selecting imbalance receiver The process to select imbalance receiver strongly depends of imbalances type, namely whether it relates an overload or an underload. The rule is that load has to be taken by a lightly loaded peer when an overload is endured. However, underloads can be handled by a lightly or heavily loaded peer. The latter is a better plan because it helps reducing overloads while fixing underloads. The selection process can leverage information previously exchanged and aggregated on peers about load (cf. paragraph explaining how imbalance is detected) in order to elect the right peer to perform relocation with. Unfortunately, when fixing overloads, there are cases where no peer which is an active member of the network respects the previous rule. For instance, all peers may be experiencing an overload. In that case, a solution is to allocate a new peer on a new machine. To enhance the allocation time, a remedy is to preallocate a pool of peers ready to join overloaded ones.

Sharing load evenly The final stage for balancing load once the type and a receiver is identified is to share as fairly as possible the resource which causes the imbalance. If the resource is balanced uniformly and if overloaded and underloaded peers perform the same, at the end the system should eventually converges to a steady state. Sharing load is strongly related to the criteria that are managed. In the following is described load sharing for our solution. Both criteria taken into consideration with our system are about quadruples and their size per RDF term. By default the CAN protocol allows to share peers' zone by splitting them sequentially per dimension (i.e. splitting a peer zone on dimension two whereas no split on dimension one was made is not allowed to avoid routing in $O(n)$) and at the middle. The issue with the middle approach is that, as depicted by

Figure 5.2(a), depending of how quadruples are mapped on the system identifier multiple splits are required. Moreover, as many peers as the number of splits are involved since by default a peer manages one zone only. Intuitively, a solution is to split at the median value instead of the middle but it has two drawbacks. First, computing the exact median value requires sorting RDF terms which is not conceivable. Second, it enables fair partitioning for our first criteria which is about the number of quadruples per peer but it leaves out our second criteria regarding RDF terms size. The right method is to compute the centroid per RDF term, this is what we do for each quadruple handled by the system. In this way, zones are split based on centroid values. The benefit is clearly identifiable on Figure 5.2(b).



(a) Cutting based on middle. (b) Cutting based on centroid.

Figure 5.2 – CAN splitting strategies comparison for sharing load about RDF data. Dashed lines depict required zone's splits.

The configuration requires with the default CAN scheme 5 splits whereas the new one based on centroid incurs 1 split only. Although cutting is enhanced, depending on how quadruples are arranged (e.g. if they are all aligned on a single dimension) up to $d-1$ splits, where d is the number of dimensions in the CAN network, may still be required to balance the resources. When the issue occurs, up to $d-1$ peers (referred to in the following as *cutting peers*) managing no resource may be injected. Since they manage no resource, these last are candidate for underload balancing in a near future which will cause additional and superfluous work to the CAN network. The situation could be easily dodged by making *cutting peers* join a same node. Even better, another solution would be to allow the peer

taking the load to manage several zones, the ones from the *cutting peers*. This way, the overhead incurred by the management of multiple peers per node is almost hidden. The former approach is probably easier to support since it leverages virtual peers whereas the latter requires changes in routing algorithms but also join and leave procedures since multiple zones are potentially managed by peers.

5.2.2 Strategies

The previous section shapes the premises for a load balancing strategy. However, a few questions are left open, especially regarding how some aforementioned parameters are set. Their definition leads to at least two main load balancing strategies, namely absolute or relative, whose details are given below.

Absolute load balancing

The absolute load balancing strategy aims to detect imbalances without exchanging information between peers. To achieve this goal threshold values are configured per criteria and passed to peers when they are deployed. These last are upper bound values that allow to signal an overload once they are exceeded. Concretely, defining such a behaviour implies to set parameters introduced with the function on Algorithm 5.1 to specific values. By setting $K_1[i]$ to 1, $K_2[i]$ to 0 and $E[i]$ to the desired threshold values, the load state estimation function works with local knowledge only. Obviously, purely local decisions has an impact on the effectiveness of the strategy, this is what we will see with the experiments.

Relative load balancing

The second strategy is about relative load balancing. Relative because local load measurements are compared to an average system load to decide whether an imbalance is experienced. To estimate the average system load measurements are exchanged between peers. Peers in charge of receiving load information depends of the gossip protocol used. As we will see with the experiments a basic strategy can be to forward load information to immediate neighbors. However, another conceivable approach is to use a mechanism similar to Mercury [24] that consists

of *a*) sending peers' load on the vicinity of each peer but also *b*) to execute periodic random walks to capture an approximate view of the global system load. Our system architecture is built atop CAN whose topology is a finite connected graph that meets the requirements of an expander graph. In other words, with a small number of edges in the network topology every peer can reach other edges by many paths. Since in [147] the authors have shown that random walks are superior to flooding in some cases of practical interest with expander graphs, applying the aforementioned gossip strategy could provide interesting results.

Optimization The gossip protocol employed could also be tweaked similarly to what is proposed in [145] by defining a parameter $p, p > 1$ so that a peer only needs to report its load if the ratio between the new load and the previous load that was sent before is greater than p. In this manner some messages are periodically evicted and peers are relieved. However, this is an additional parameter that affects the convergence time of the algorithm and it would require intensive benchmarks.

Upstream load balancing

Upstream load balancing is a strategy that complements the ones introduced previously. In 3.2.2 we explained that quadruples are routed according to their RDF term values. The idea with upstream load balancing is to index quadruples by considering their RDF terms once they have been passed to a function that applies a transformation to improve skewed distribution due to common prefixes. Such a function could consist of removing for instance the namespace from IRI values associated to RDF terms. Therefore, if we have several quadruples with predicate values that share the same namespace, the information is indexed by keeping the predicate local part only. To better understand, let us consider the Dublin Core [148] vocabulary. It defines an ontology with multiple metadata for describing resources in their generalities. This vocabulary contains for example the predicates *dc:title* and *dc:creator* where *dc* refers to the standard Dublin Core namespace[2]. The removal operation consists in cutting out the characters until the last / or # character. Thus, *dc:title* and *dc:creator* are indexed lexicographically

[2] `http://purl.org/dc/elements/1.1`

by using respectively *title* and *creator*. As the last two values differ from the first character whereas the old values differ after the first thirty two characters, they have much more chance to be indexed on two different peers than before. However, this probability highly depends on the CAN space splitting and the RDF data to index. Although this scheme brings a few improvements it cannot deteriorate load balancing. Similarly to prefix removal, another solution may be to compare during request routing the bounds of a zone managed by a peer and the RDF terms values based on the reverse RDF terms value so that skewed prefixes less impact data placement on peers. The advantage of this second approach compared to prefix removal is that it incurs no overhead since it may be implemented by redefining how the function that performs the comparison during routing behaves. However, both methods must be used when exact matching only is required with subscriptions or synchronous SPARQL queries because they incur flooding to find all matching results when filter constraints are specified. Consequently, this last solution should be enabled as a complement to one of both methods (*prefix removal* or *reverse comparison*) presented previously when the usage context is well defined.

5.3 Evaluation

The experiments presented in this section have been restrained to some key evaluations. Load balancing is assessed for a network configuration initialized with one peer. Furthermore, only overloads are considered by setting parameter $K_2[i]$ for the decision function to 0. Once detected, imbalances are corrected by making new peers, deployed on dedicated and preallocated machines, to join the ones that are heavily loaded. Load balancing involving relocations is let for future work.

In contrary to evaluations made in previous chapters, the following experiments rely on real data extracted from a Twitter[3] data flow by writing an adapter in Python. The workload is about 10^4 CEs, each embedding 9 quadruples. Since one meta-quadruple is automatically generated per CE, at the end the P2P network handles 10^5 quadruples. To allow reproducibility but also to make a fair comparison between the different strategies that are evaluated, the same workload

[3] `https://dev.twitter.com/docs/streaming-apis`

is reused in all the experiments.

The first assessment is about the overhead induced by statistical information recording. Figure 5.3 depicts the time required to acknowledge the insertion of quadruples when statistical information recording is respectively disabled or enabled. Each result is the average time from five runs whose the first two are let aside due to JVM warmup. Computing the mean or the centroid in the same thread increases the overall insertion time by approximatively 3.45. The overhead is explained by the fact that RDF term values are converted in radix 10 and back to 1114112 (the radix associated unicode characters) for computing online mean or centroid values. To address the issue, a thread pool is introduced. By using a hard drive disk our experiments have shown that two threads are required to hide the overhead induced by the computation of statistical information. Finally, estimating the mean or the centroid makes almost no difference.

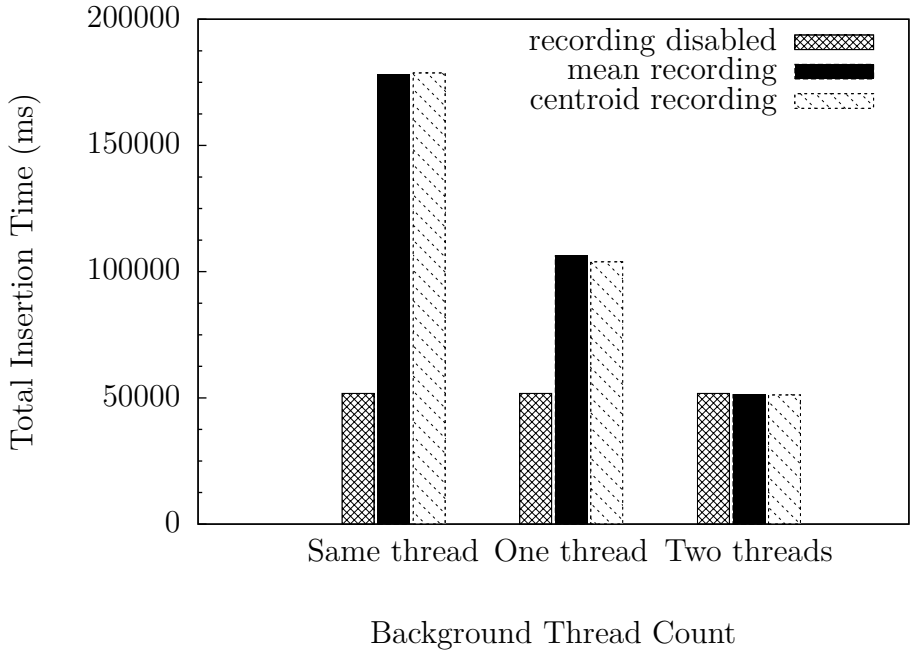


Figure 5.3 – Statistical information recording overhead.

Before evaluating the absolute and relative strategies, we have performed an experiment to see what could be the best distribution. The scheme consists of injecting the workload on a single peer and once all quadruples have been stored

to start load balancing iterations. Each load balancing iteration consists of picking a new peer from the preallocated pool of peers and to make it join the one from the network that is the most loaded, thus simulating an oracle. The action is repeated until to have a network containing 32 peers. To show the interest of using statistical information, the experiments have been performed, as depicted on Figure 5.4, by using zones cutting based on their middle or centroid values recorded on the fly. By applying the default CAN rule, which cuts 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 that have their load close to the ideal distribution. Although the distribution is not perfectly distributed, it is greatly improved.

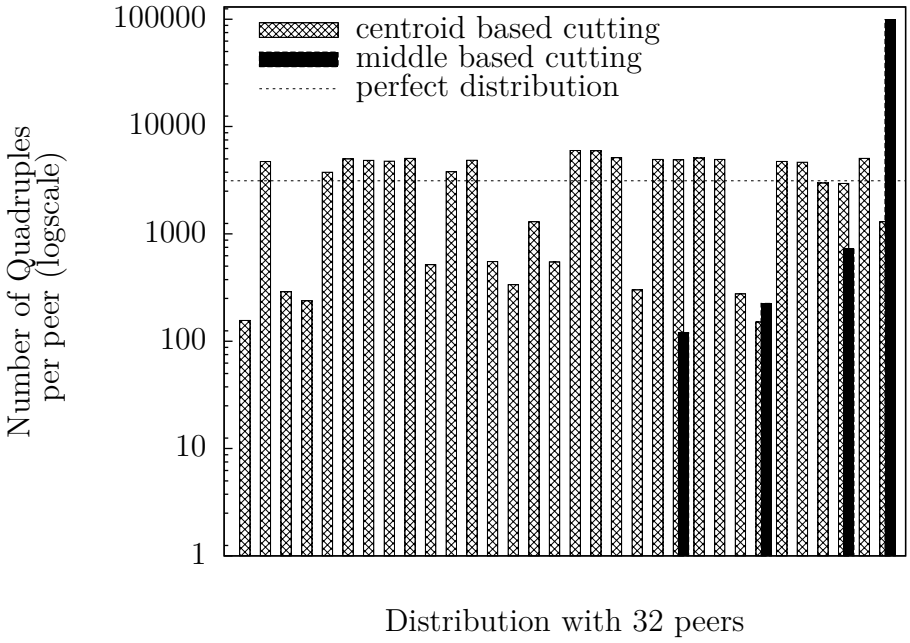


Figure 5.4 – Static load balancing using middle vs centroid partitioning.

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 deviation with the mean times 100. In the following we use this estimator to compare strategies. For information, the coefficients are 559.4% and 69.5% when the middle

and centroid methodologies are respectively applied with the static load balancing experiment presented above, thus showing that centroid performs better because this last value is eight times lower than the previous.

Finally, we have compared the absolute and relative strategies. For the absolute one, threshold value is set to the number of quadruples divided by the final number of peers, which gives 3125. The relative strategy does not rely on global knowledge, and $K_1[i]$ was set to 1.1 so that overload is detected when local measurements on peers is greater than or equals to 1.1 times the estimate value computed by receiving load information from immediate neighbors. The parameter K_1 was set according to previous experiments that let suppose the best distribution is achieved for this value. Table 5.1 shows the results obtained according to the strategy applied and put it in correlation with the results got for the static load balancing experiment that exhibits the best distribution that can be achieved according to the relative standard deviation (69.5%). In summary, 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 static load balancing solution but it achieved a better distribution (96.57%) than the absolute strategy and this without using global knowledge. Besides, since more peers are receiving RDF data, more nodes are involved to answer subscriptions with the pub/sub layer, thus increasing the throughput in terms of CEs received per second.

	Static load balancing		**Dynamic load balancing**	
	Middle	*Centroid*	*Absolute*	*Relative*
Relative stddev	559.4%	69.5%	119.75%	96.57%

Table 5.1 – Load balancing strategies comparison.

Summary

This chapter has presented and briefly analyzed two strategies for balancing RDF data on our revised CAN network. The central idea is to share overloads between

peers by splitting peer zones not at their middle as suggested by the default CAN protocol but at the point that fairly balance RDF data. This is made by recording centroid values of RDF terms per dimension. Then, the strategies that are proposed mainly differ regarding how imbalances are detected. The first uses global knowledge whereas the second relies on information exchanged between peers. Experiments have shown the latter strategy performs better than the former. Although the solution we propose is far from ideal in the sense that RDF data are not as well distributed as it could, the strategies enhance the distribution on peers, the involvement of peers and thus the throughput when publish/subscribe is used.

It is worth mentionning the presented solution is an unfinished work and many points would require more intensive investigations and experimentations. Also, many faces of our work could be enhanced. For instance, the gossip protocol to use would require refinements by implementing optimizations, relative to load dissemination, proposed in Section 5.2. Furthermore, before allocating new peers, relocation should be envisaged. One additional direction is to consider more criteria such as queries load, subscriptions or even CPU and bandwidth consumption. Since our balancing model has been designed with the idea to support multiple independent criteria, adding new ones should not be arduous.

Chapter 6

Implementation

Contents

In this chapter we introduce and give details about the implementation of the EventCloud (EC) middleware which is the software used to assess the different algorithms and features which have been presented in the previous chapters. First, we start to review the software architecture associated to the EventCloud middleware by focusing on its modularity and the key software elements that interact together. Then, in a second time we discuss how we have tuned performances by applying different methods at several levels for the different performance bottlenecks which have been identified throughout the development of the middleware. It includes a custom cache layer but also serialization and MAOs improvment.

6.1 Middleware Design

The EventCloud middleware has been designed from scratch and implemented by using the ProActive Programming middleware. This gives us the opportunity to build the whole system on concepts and abstractions that may be easily reused, extended or replaced. In contrary to an integrated architecture, where no clear separation exists between software elements, here key elements are identified. Given that the implementation is in Java, we make an intensive use of Maven[1] to manage the lifecycle of the project but also to keep the architecture structured and modular by associating multiple modules to the system elements, thus isolating functionnalities. This way, only the modules that are required may be loaded. Besides, the modular architecture and the manner to proceed allows to replace or add any software element or module without affecting the rest of the system.

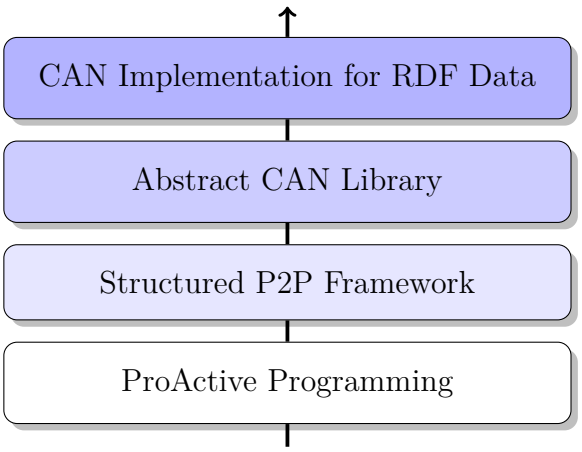


Figure 6.1 – Stack of main software blocks designed and/or used. Colored blocks have been designed from scratch. The color intensity depicts also the level of abstraction.

The Content Addressable Network protocol is at the core of the EventCloud infrastructure. As depicted by the Figure 6.1, to make the architecture as modular as possible, the CAN network that is used with the EventCloud is defined on top of a stack made of four main software blocks: the existing ProActive Programming middleware at the bottom which abstracts communications, the structured

[1] `http://maven.apache.org`

P2P framework above that provides reusable concepts to design structured P2P networks and then just below the EventCloud specific CAN implementation, an abstract CAN implementation that provides the subsistence minimum to connect peers and route messages in a CAN topology. The following sections discuss in details how these main software blocks have been designed and implemented.

6.1.1 A generic structured P2P framework

The generic structured P2P library is at the core of the middleware we have designed. This first block is a bunch of interfaces and classes that provide the necessary abstractions and reusable concepts to ease the implementation of structured P2P protocols with the ProActive Programming framework.

Peers representation

Figure 6.2 depicts a simplified version of the classes that describe the structure of the generic library for creating a model of a peer. The first step to design the library was to identify the key operations that are common to any structured P2P protocol. The *Peer* interface defines the signature of these operations that allow basic interactions such as creating a network, joining another peer, leaving a network or sending operations and routing messages as we will see later. However, a peer implementation is not specific to a protocol. It is mainly used to exhibit the common operations remotely and to manage the state of a peer (i.e. whether it has already joined a network or not). Thus, the second step consists in deciding how a peer is specialized for a given network. In object oriented programming, composition or inheritance may be used. However, it is acknowledged that composition is usually more flexible than inheritance and allows the container class to be more stable in the long term [149]. For this reason, common structured P2P operations are delegated to a *StructuredOverlay* by means of composition. This last is an abstract class maintaining attributes and methods that are still commons to any structured P2P network but that should not be made public to external users since they are useful for protocol implementers only. Besides, a *StructuredOverlay* implements a *DataHandler* interface that abstracts the storage on peers. The interface defines only a few methods that allow to assign data but

also to retrieve or remove data according to a space managed by a peer during a join or leave operation. A protocol implementation is made by using the provided abstractions and creating a concrete class that inherits from *StructuredOverlay*, such as a *CanOverlay* or *ChordOverlay* implementation. The manner to proceed is in fact an application of the Strategy design pattern that allows to select an algorithm's behaviour at runtime. This way, the implementer may rely on existing methods and values from *StructuredOverlay*, override methods to further improve features that are then tasted by users with polymorphism.

Now that peers may be defined for different protocols, we have to determine how they are instantiated and disclosed remotely in order to interact with others. In ProActive, remote entities are represented by Active Objects or GCM components. We have chosen to associate a peer to a GCM component because components allow a clear separation of functional and non-functional concerns while improving reusability, which is an interesting property for adding monitoring in the future.

Communications between peers

P2P networks are designed to share resources among interconnected peers and resources are made available to others or looked up by enabling communications between peers. In structured P2P networks, peers are organized in a structured way so that they manage part of a global identifier space. Consequently, there is at least two manners to point out a peer, either its remote reference is known and it may be contacted directly, or only a key that is contained by the space interval one ore more peers maintain is specified. In this last case, routing is required to find out the final destination. We detail hereafter how our generic P2P library is designed to support both communication types.

Operations Operations are special messages that are sent by invoking remotely the *receive* method on a *Peer* stub. Their purpose is to hide operations that should not be exposed to users while keeping the *Peer* interface untouched. Operations are declined in two flavours, *RunnableOperation*s that return no result and *CallableOperation*s that give back a *ResponseOperation*. Both may be invoked asynchronously since *ResponseOperation* is reifiable. Beyond that, the two operation classes extend an abstract *Operation* class that can be used to implement for

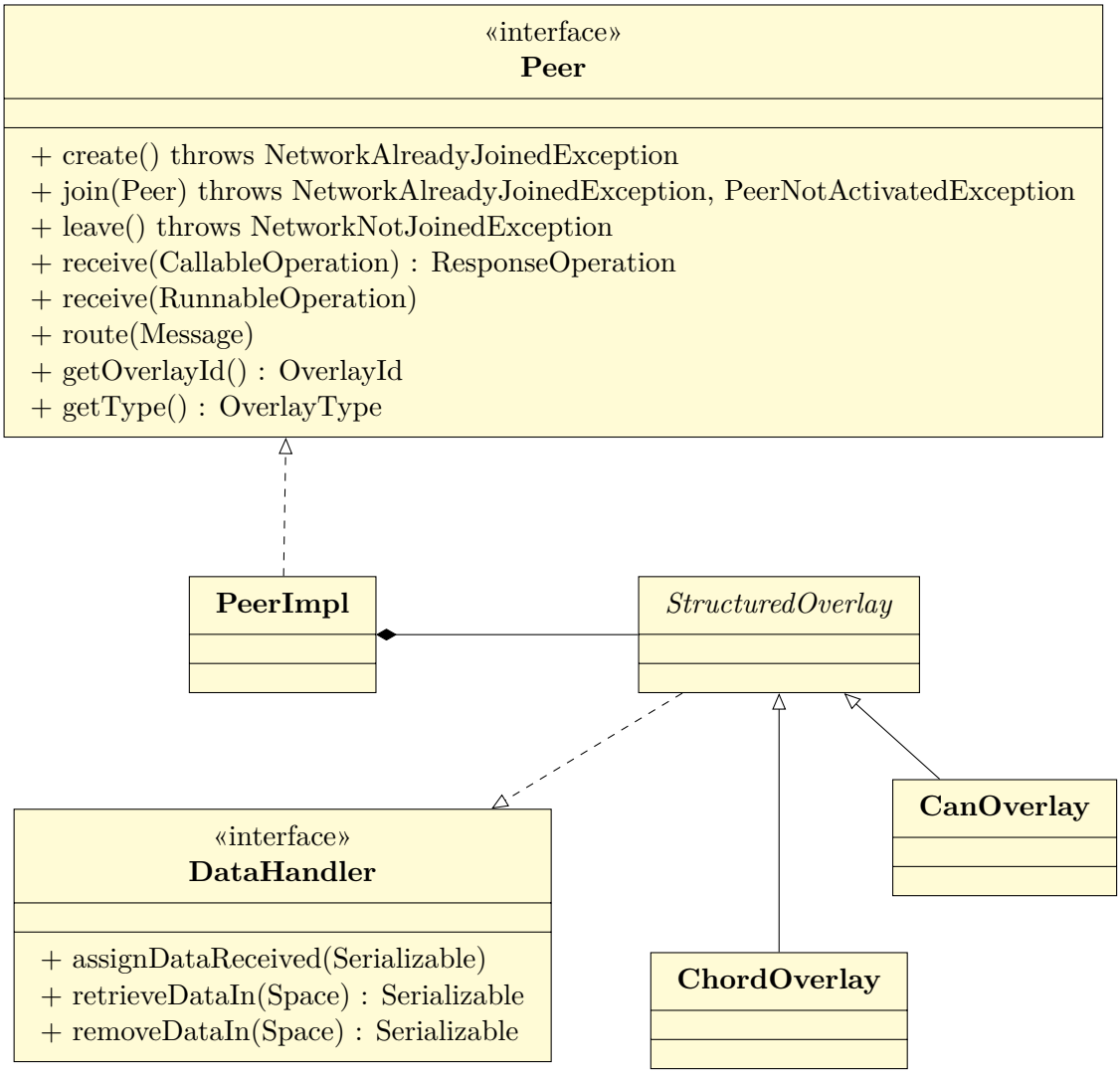


Figure 6.2 – Simplified version of the class diagram defining a peer.

instance permission checking or even to define compatibility with others requests as we will see in Section 6.1.1.

Messages A message is a specific envelope that contains an information to deliver or an action to execute on a set of peers according to a key. Figure 6.3 shows the class diagram describing the model for messages and the abstractions required for routing them. Messages are essential elements of the whole architecture. They are modeled as an abstract *Message* class. This class contains common attributes such as an identifier, a key that is used to know whether a message received on a peer has to be delivered or not, but also others instance fields that maintain statistics (e.g. the number of hops), which is particularly useful to assess the performance of a routing algorithm. The *Message* class is subclassed with two child classes which are the *Request* and *Response* classes. The reason lies in the fact that requests and responses do not share the exact same behaviour and states. The former saves values about the time at which the request has been dispatched whereas the latter keeps the number of hops related to the routing of the response only. However, a *Response* references a *Request* when it is built to have access to the request attributes that are necessary to compute some measurements once a response has been delivered (e.g. the round-trip delay time). Not all requests necessarily generate a response. The distinction between requests that yield responses and those that do not is made with the help of the *responseProvider* instance field that contains an instance of the *ResponseProvider* class which describes if a response has to be created and how.

In both cases a *Message* is routed according to a *Key*. Routing is performed from a peer by calling the one-way *route* method. The call to the *route* method results to an invocation of the same method name on the embedded overlay object, that itself delegates the routing to the initial message instance that is routed, by double dispatch [150]. This way, the routing decision is taken at runtime by the concrete message's type that knows which kind of router to use. The router that is used to route a message is abstracted through a *Router* interface that defines a public method named *makeDecision*. This method is executed each time a message reaches a new peer. It takes as parameters the message object and the overlay on which the message has been received to know whether the destination is reached

or if further routing is required. Besides, decoupling a *Router* from a *Message* allows different messages to reuse existing routing algorithms. For instance if we define a multicast algorithm for CAN that uses a multicast key to make the routing decision, this specific router could be used to route messages whose the purpose is to retrieve quadruples but also for those that aim to remove quadruples since only the action to perform once the destination is reached differs. This behavior that is specific to any *Message* is defined by overriding the *onDestinationReached* method.

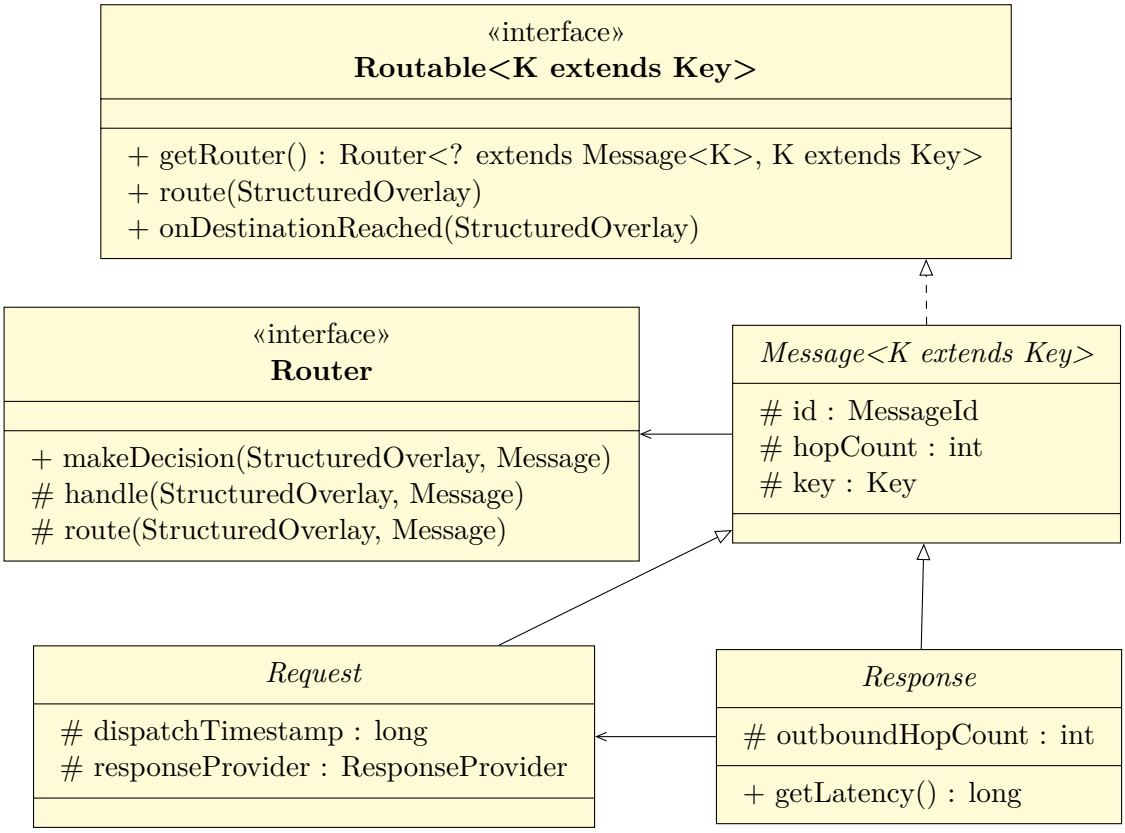


Figure 6.3 – Simplified version of the class diagram defining messages.

Bootstrapping

In overlay networks, bootstrapping usually refers to the procedure to discover peers that are already member of a running P2P network. To enable peers' dis-

covery, references to peers that are part of a P2P network are retained by trackers. Trackers are GCM components organized into a fully connected mesh network that forms an entry point to a P2P network. The view is maintained between all the trackers with the help of the collective group communication feature provided by ProActive [151]. Compared to an rmiregistry, a tracker is a protocol agnostic registry.

It may be argued that fully connected networks are not scalable, however trackers are assumed to be deployed, in our context, on stable nodes and peers on a trusted infrastructure such as a datacenter. Moreover, peers' references are lightweight objects and trackers are supposed to track only some references to peers from a P2P network, not all. The idea is to always have access to one reference in order to interact with a running network. Obviously, there is a tradeoff between the number of references that are kept and how operations are initially balanced on the P2P network since the less references are available, the less entry points can be used. For this reason, references are saved according to a configurable probability. To enforce the storage probability, *create*, *join* and *leave* operations on peers are delegated to trackers. To insert a peer into a P2P network, an *inject* method is called on a tracker with a peer stub. This last method allows to call the *create* or *join* method on the specified stub, depending on whether the peer is the first to join the network or not. Moreover, the method *inject* ensures that the peer reference is saved if it has to be while multiple injection strategies may be supported. Other methods are remotely accessible such as a *takeout* procedure that applies a similar behaviour as the *inject* one but for asking a peer to leave the network. Finally, to have the possibility to discover references, *getPeers* and *getRandomPeer* methods are provided by trackers. In case trackers' scalability is an issue, they could be organized in a structured P2P network such as Chord. However, our experiments showed this is not be required on trusted environment with an acceptable number of peers.

Although trackers are deployed on stable nodes, IP addresses are not easy to remember, especially with the advent of IPv6. A solution could be to rely on the Domain Name System (DNS). By associating an address record to the IP addresses of trackers, a set of tracker can be identified through a simple and easy to remember domain name. This way, users benefit also of the load balancing

property applied during the resolution of domain names, which allows to balance load among trackers with an associated IP address declared in the records.

Proxies

We have seen in the previous section that trackers are entry points to a network created with our abstractions. When a user, an external application, or more generally an entity wants to interact with a P2P network, it has to know at least a domain or a subdomain name that once resolved, points to the IP address of one tracker. Afterwards, the entity has to contact this tracker in order to get back a peer reference that can be used to route for instance a message. To prevent users from this off-putting sequence of actions we have introduced the notion of proxies. A proxy represents a gateway between a P2P network and an entity that wish to collaborate together. As depicted by Figure 6.4, a proxy offers multiple methods to forward a *Request*, but also a method to receive responses. The reason lies in the fact that all communications made in the P2P network are one-way, as we explained previously. When the *sendv* method is used, the dispatching is delegated to a *MessageDispatcher* which simply forwards the request to a peer after having retrieved its reference from a tracker. However, others *send* methods that entail responses require more processing. In that case, the *MessageDispatcher* instance forwards the request, as for the *sendv* method, but waits for a response. The synchronization point is created on the proxy by suspending the thread in charge of sending the request with the help of the Java monitors. The suspended thread wakes up when the response arrives. The correlation between a request and its response is made based on a unique identifier set on each request before being dispatched. It is worth notice that synchronization points for requests with responses could have been managed transparently by using ProActive futures. However, futures entail to use the same path for a request and its response, thus forcing to cross unnecessary peers with potentially large payloads. For instance, a request with an unicast constraint that does not require to retrieve back information on peer crossed during the routing may have its response sent directly from the peer handling the request to the proxy.

The *send* method that takes three parameters, and whose the first is a list

of requests, is particularly useful to dispatch multiple requests whose responses should be merged together before being delivered to users. The idea is to combine responses at the edges of the P2P network to reduce the final response size and as a consequence the time required to transfer the result from the P2P network to proxies. The merge operation is defined through a *ResponseCombiner* by using a *Context*.

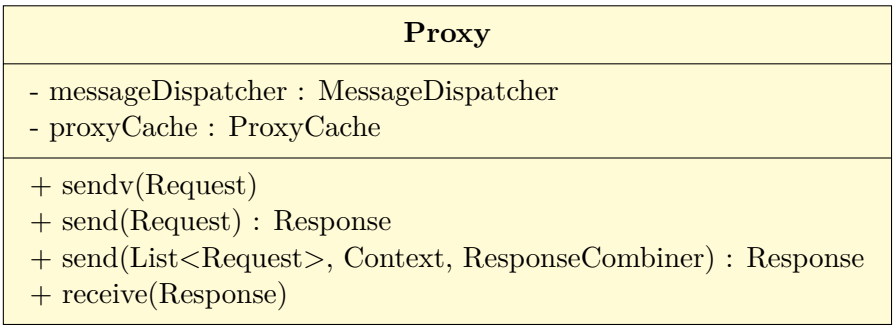


Figure 6.4 – Simplified version of the class diagram defining a proxy.

Beyond to ease interactions, a proxy has multiple advantages. It shifts synchronization points from the P2P network to users which prevents deadlocks in case of churn or failure. Additionally, a proxy features a cache mechanism that prevents contacting trackers each time a request has to be dispatched, thus improving the routing latency. Finally, proxies may be used to provide enhanced filtering capabilities outside a P2P network or to enforce end-to-end properties.

To better explain how a proxy behaves, Figure 6.5 depicts a sequence diagram that shows how objects introduced so far operate with one another and in what order for sending a request and routing back a response. Let's assume an entity e whose its desire is to dispatch a request req that generates a result through a response res. First, e invokes the *send* method on a proxy instance with the request object req. As aforementioned, the first action to perform is to get a reference to a peer that is an active member of a P2P network. To do so, the proxy contacts its *ProxyCache* instance c that aims to keep in local references to active peers. The first time it is contacted, the cache store is empty and has to be populated. The action is achieved by calling the remote method *getPeers* on a tracker. Peer references that are returned are stored in the *ProxyCache*. Then, one among those available is picked at random and used to forward the request on the P2P network.

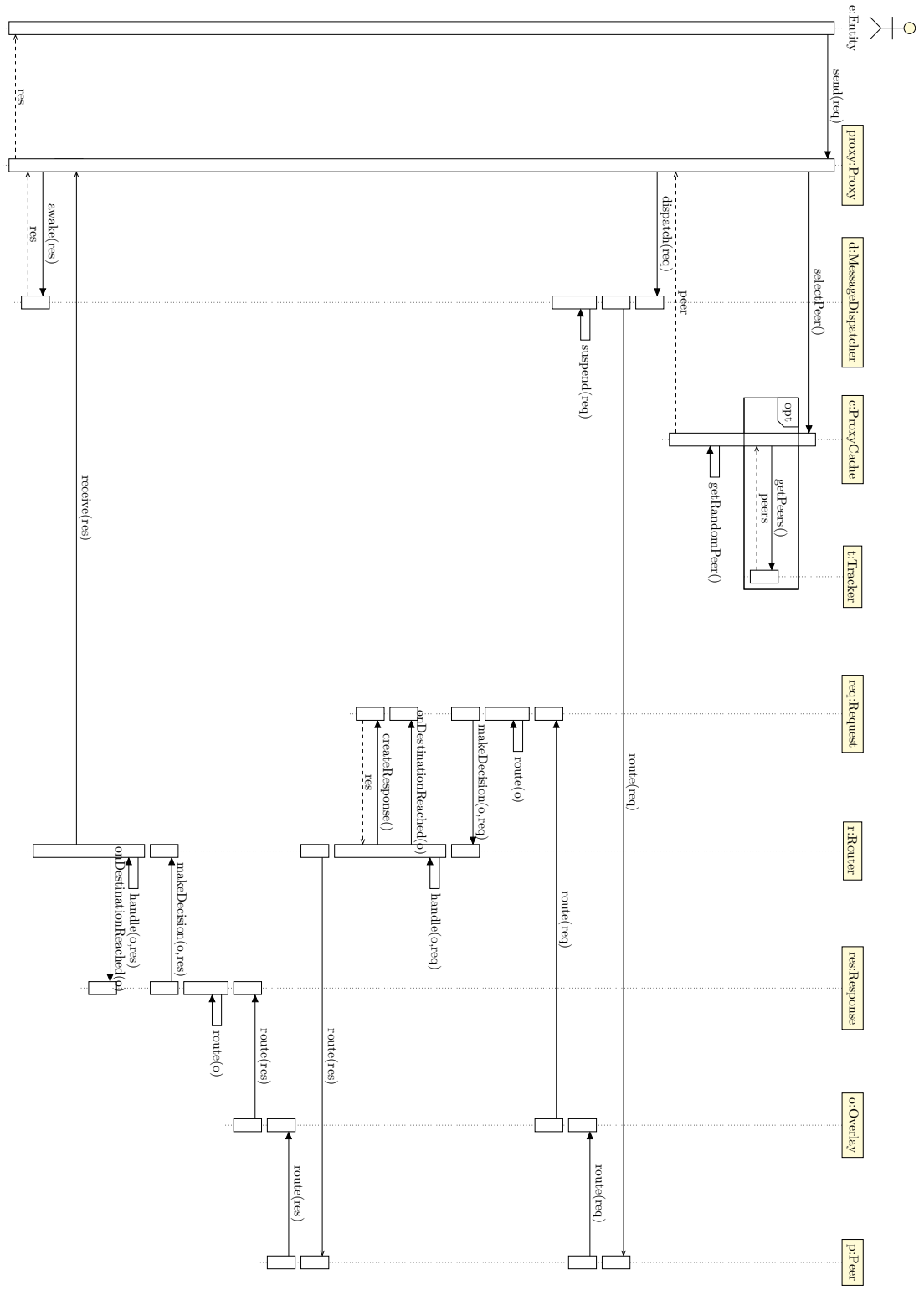


Figure 6.5 – Sequence diagram showing a proxy interaction to send a request and to receive a response on a P2P network modeled with the proposed abstractions.

Futures interactions with c result in local actions only. The dispatching of req is made by a *MessageDispatcher* that knows how to dispatch requests. Internally, it stores in a table an entry for the request id for which a response is expected, then it forwards the request to a peer by invoking the asynchronous *route* method. That, before suspending the current thread with a Java wait call on the entry which has been previously stored. The peer that receives the call to route the message req delegates it to its concrete overlay implementation that itself brings the routing decision to the runtime message type that knows which router to use. The action is done by double dispatch to obtain the overlay and message instance as parameter of the method *makeDecision* executed on a router. From the parameters, the router is able to decide whether the message requires further routing or not. Let's say that we have only one peer that manages the key carried by the message, then, the user action is executed with a call to *onDestinationReached* and a response is created by using the *ResponseProvider* contained in the req object. Once the response is created and attributes are set, it is routed back. Similarly to requests, the routing is fully abstracted and how routers operate is consequently application and use case dependent. It may be decided to use the reverse forwarding path or to send back the answer to the proxy in one hop. The response eventually reaches the proxy with an invocation of its remote method dubbed *receive*. Upon its execution, the method retrieves the entry, attaches the response to the entry and wakes up the thread that was waiting the response with a Java notify call on the entry. That way, the sending thread is awaken, retrieves the response from the entry which has just been updated and returns res to e. It may be observed that a proxy must be remotely accessible to receive responses routed in one-way, for this reason a proxy is a remote object. It is designed as a GCM/ProActive component using the multi-active objects extension similarly and for the same reasons as peers.

Configuring multi-active objects

Working with multi-active objects (cf. Section 2.4.2) requires three main actions:

1. Identifying whether a *soft* or *hard* limit must be used;

2. Defining groups and compatibilities in order to improve parallelism by handling when possible some requests in parallel to others;

3. Evaluating empirically the correct value for the limit defined previously.

We are now discussing about the type of the limit used for the various remote objects introduced with the generic structured P2P library. Up to now three kinds of remote objects have been introduced: trackers, peers and proxies. All three are GCM/ProActive components that make use of a multi-active serving policy. For trackers we use a hard limit because no re-entrant call are performed. However, proxies and peers handle re-entrant calls, either to dispatch messages or by handling operations through a join or leave request for instance. Therefore, both rely on a soft limit. The values set for the different limits are discussed in Section 6.2.1 since their definition fall within performance tuning.

Although proxies define a soft limit, deadlocks may still occur when requests with responses are dispatched. The reason lies in the manner threads are suspended to await responses. It is done by using java monitors, which bypass the multi-active object library. Consequently, the request executor is not aware that a thread has been suspended and it considers a sleeping thread as running, which leads to inconsistent states when a decision to schedule requests is made. A solution could be to use futures and wait-by-necessity but it is not a viable solution as we already explained. Thus, to address the issue two new methods have been added to the multi-active objects API so that it is possible to decrement and increment manually the number of active threads.

Regarding the definition of multi-active groups and compatibilities, it remains simple for trackers and proxies. With a tracker, methods that are used to retrieve peer references are members of a group named *parallel*, which declares to be compatible with itself so that all methods in this group can be executed simultaneously. Other methods used to inject and takeout peer references, but also the one used to connect a tracker to another, do not declare a group membership. This way, they are executed in FIFO order and in mutual exclusion with other methods from the *parallel* group. Similarly, a proxy defines a *parallel* group and assigns all its remote methods to this group. In that case, to achieve mutual exclusion, the reponsability is handed over to the programmer. The reason lies in the fact

that not all the method requires a full synchronization. Besides, the management of this specific case may be handled easily and efficiently with a finer granularity. Thereby, thread-safe data structures are used to prevent race conditions when requests and responses are respectively dispatched and received through an instance of *MessageDispatcher* which maintains awaited message identifiers.

Compatibility definition is more complex for peers. That's because a peer exhibits remote methods whose execution differs from a structured P2P protocol implementation to another. For instance, the compatibility between two operations received on a peer may depend of the concrete operations type but also the peer state. Fortunately, the multi-active object library allows to decide about compatibility at runtime with the help of compatibility functions (cf. Section 2.4.2). Listing 6.1 shows the groups that are defined along with the compatibilities that are loaded to the framework for a peer implementation. Six groups are declared (line 2–11). The first group dubbed *readImmutableStateOnly* is used to serve in parallel to others, and itself, all requests that simply access or read and return an immutable field value. For example, the methods *getOverlayId*, *getType*, *equals* or even *hashCode* from the *Peer* implementation are member of this group. This first group can be considered as a rule of thumb when dealing with multi-active objects. Then, the next two groups (*join* and *leave*) are associated to the respective methods of the same name. Finally, *receiveCallableOperation*, *receiveRunnableOperation* and *routing* are associated to their respective methods in the *Peer* interface, namely the methods *receive(CallableOperation)*, *receive(RunnableOperation)* and *route(Message)*.

Although join and leave requests are not compatible with themself, groups definition is required to define what is their compatibility with messages and operations. The reason lies in the fact that a join may require to update neighbors and to wait for a reentrant call with the reception of an operation. Similarly, some messages, such as monitoring messages may be executed while a join request is being served.

Then, after the definition of groups, compatibilities between groups are declared (line 13–40). It is done for each possible combination of the groups by using the *@Compatible* annotation. The interesting point here is that we have introduced a specific *condition* parameter. This parameter allows to specify the name

of the function to use for deciding whether two groups are compatible. To define a compatibility function that depends of a peer state, the name of the compatibility function is prefixed with the *this.* keyword. This way, when the multi-active objects framework attempts to check compatibility for two methods that belong to the groups associated to the condition, the compatibility function is searched in the peer implementation which allows to access to the instance fields it contains, these last characterizing its current state. In our case, each compatibility function local to a peer has its implementation delegated to the *StructuredOverlay* instance embedded by the peer so that the value returned is defined per overlay implementation and is not common to all P2P protocols. When the compatibility depends of the parameters only, the prefix *this.* is ommited and the function is looked up in the class specified with a group-parameter (line 9–10).

Finally, the *routing* group is defined as self-compatible. We have made this choice because most of the time routing implies to access data structures that are not updated at the same time. The only issue to care about is when a message reaches its final destination and that an action is executed. Since, two messages may reach the same peer at the same time, the same action or two actions that touch the same data structures or counters may be executed at the same time. When this issue occurs, it is up to the programmer to synchronize what is required. As a result, only part of the routing process requires synchronization.

```
1  @DefineGroups({
2    @Group(name="readImmutableStateOnly", selfCompatible=true),
3    @Group(name="join", selfCompatible=false),
4    @Group(name="leave", selfCompatible=false),
5    @Group(name="receiveCallableOperation", selfCompatible=true,
6           parameter="org.ow2...operations.CallableOperation",
7           condition="this.areCompatible"),
8    @Group(name="receiveRunnableOperation", selfCompatible=true,
9           parameter="org.ow2...operations.RunnableOperation",
10          condition="isCompatible"),
11   @Group(name="routing", selfCompatible=true)})
12 @DefineRules({
13   // readImmutableStateOnly is compatible with all other groups
14   @Compatible(value={"readImmutableStateOnly", "join"}),
```

```
15    @Compatible(value={"readImmutableStateOnly", "leave"}),
16    @Compatible(value={"readImmutableStateOnly",
17                       "receiveCallableOperation"}),
18    @Compatible(value={"readImmutableStateOnly",
19                       "receiveRunnableOperation"}),
20    @Compatible(value={"readImmutableStateOnly", "routing"}),
21    // callable operations compatibility
22    @Compatible(value={"receiveCallableOperation", "join"},
23                condition="this.isCallableCompatibleWithJoin"),
24    @Compatible(value={"receiveCallableOperation", "leave"},
25                condition="this.isCallableCompatibleWithLeave"),
26    @Compatible(value={"receiveCallableOperation", "routing"},
27                condition="this.isCallableCompatibleWithRouting"),
28    // runnable operations compatibility
29    @Compatible(value={"receiveRunnableOperation", "join"},
30                condition="this.isRunnableCompatibleWithJoin"),
31    @Compatible(value={"receiveRunnableOperation", "leave"},
32                condition="this.isRunnableCompatibleWithLeave"),
33    @Compatible(value={"receiveRunnableOperation", "routing"},
34                condition="isCompatibleWithRouting"),
35    // callable and runnable operations are jointly compatible
36    // under some conditions
37    @Compatible(value={
38                  "receiveCallableOperation",
39                  "receiveRunnableOperation"},
40                condition="this.areCompatible")})
41  public class PeerImpl extends AbstractComponent
42              implements PeerInterface, PeerAttributeController {
43      // ...
44  }
```

Listing 6.1 – Groups and compatibility definition using multi-active objects annotations on a Peer implementation.

6.1.2 An abstract CAN library

The second software block concerns the abstract CAN library. It is designed by reusing the generic structured P2P framework. Protocol specific features are

defined in an abstract *CanOverlay* class that inherits from *StructuredOverlay*. A CAN overlay is mainly characterized by the zone and the neighbors it stores.

The neighbors are organized in a data structure per dimension and direction. That's way neighbors of interest may be quickly identified, updated and removed during a join, leave or route operation. The zone definition has also required some attention to support multiple representations of its elements. A zone or hyperrectangle in a d-dimensional Cartesian space may be represented by an upper and lower bound. These bounds are usual points made of coordinates. However, points may be depicted as numeric, alphanumeric or even Unicode values. The last being the solution we have adopted. To make the coordinates independent of their representation, an abstract class defines common methods such as one that splits a coordinate in its middle, or others that are used to compare coordinates and points together. Then, multiple coordinate implementations may be provided. By default, only one for coordinates modeled as floating point values in radix 10 is provided.

The routing algorithms, introduced in Chapter 3, based on unicast and multicast constraints have been implemented at this level. Although both versions of the multicast routing algorithms have been implemented (the naive and the optimal one), the optimal based on broadcast is enabled by default since it avoids many duplicates.

Finally, it is worth mentioning that the different features that are provided at this level are designed to work for any number of dimensions a CAN network is supposed to be instantiated with.

6.1.3 A CAN implementation for RDF data

The upper software block (cf. Figure 6.1) corresponds to the concrete implementation of our EventCloud middleware. Figure 6.6 sketches a high-level view of the architecture. It involves an extended version of the software blocks and thus elements which have been described above, namely peers and proxies. Beyond trackers that maintain references to some peers and serve as the entry points to a 4-dimensional Content Addressable Network, the structured P2P network is made of peers. As we explained in Chapters 3, 4 and 5, peers are in charge of storing

RDF data, subscriptions, to perform the matching between both but also to assist with the resolution of SPARQL queries while keeping data well balanced. To achieve this purpose their internal is designed with multiple software elements (i.e. artifacts) whose each has its own role.

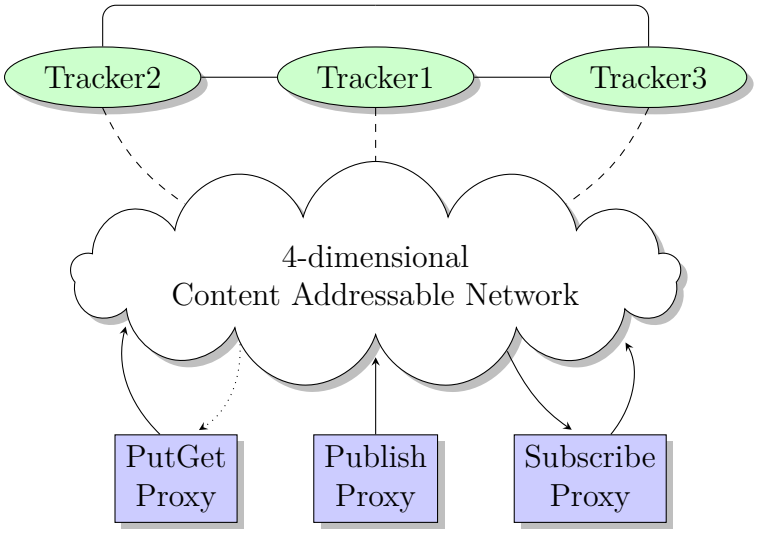


Figure 6.6 – High-level view of the EventCloud architecture.

Proxies, which allow interactions with the CAN network from external entities (i.e. applications or users), are proposed in three flavours: *PutGet*, *Subscribe* and *Publish*. PutGet proxies allow the addition and the removal of RDF data but also the retrieval of semantic information through the distributed execution of SPARQL queries based on the traditional and synchronous query/response model. Subscribe and publish proxies leverage the asynchronous publish/subscribe communication style. The first is used to submit subscriptions and to deliver notifications that are matching whereas the second is used to publish new information to the brokering network. Similarly to peers, dedicated software elements are defined and reused between proxies.

In the next sections we discuss proxies and peers internals. We will further see that one of the goals when designing the middleware was its modularity (i.e. to be able to easily change, modify, replace or reuse some parts).

EventCloud proxies

Proxies at the EventCloud level reuse by composition the proxy abstraction introduced in the Section 6.1.1 since all proxies require to interact with a CAN network by sending messages. However, each EventCloud proxy behaves differently and exposes a different API. Public APIs are sketched with Figure 6.7. The important point to notice here is that we do not make use directly of an existing RDF Java API for RDF specific abstractions (e.g. Quadruple, SPARQL query results, etc.). Instead, we introduce our own RDF abstractions, even if then the abstractions we provide are bound to objects from a standard RDF API, the native one provided by Jena. The reason to do so lies in the fact that, at the time we started to design our system, most of the existing RDF API were not serializable which is really not handy to work with in a distributed context. Moreover, some RDF abstractions such as the quadruple one requires to store publish/subscribe specific values like the time at which a quadruple has been published, which is again not supported by existing APIs. Additionally, SPARQL query responses may contain extra information related to the infrastructure where the execution took place such as the number of hops required, the execution time, etc. Besides, the added value to expose to users our own RDF abstractions is that the dependency to an RDF API (such as Jena [152] or Sesame [78]) is isolated in our abstraction. Thus, moving from one to another just consists in plugging a new implementation of our abstractions. In the past, we have successfully swapped from Sesame to Jena without impacting the other parts of the architecture.

Some may argue we could use RDF2Go [153] which is an abstraction over triple and quadruple stores that allows developers to program against RDF2Go interfaces and choose or change the implementation later easily. However, RDF2Go interfaces are not serializable, maintained sporadically and the overhead induced by the library was not negligible at the time we tested it. Therefore, we have put this solution aside.

Internally, proxies share common software elements. Figure 6.8 depicts what are those that are reused between proxies. For instance, with a *PutGet Proxy*, the execution of a SPARQL query involves three software elements, namely a *QueryPlanGenerator*, a *QueryDecomposer* and a *QueryPlanExecutor*. The first is

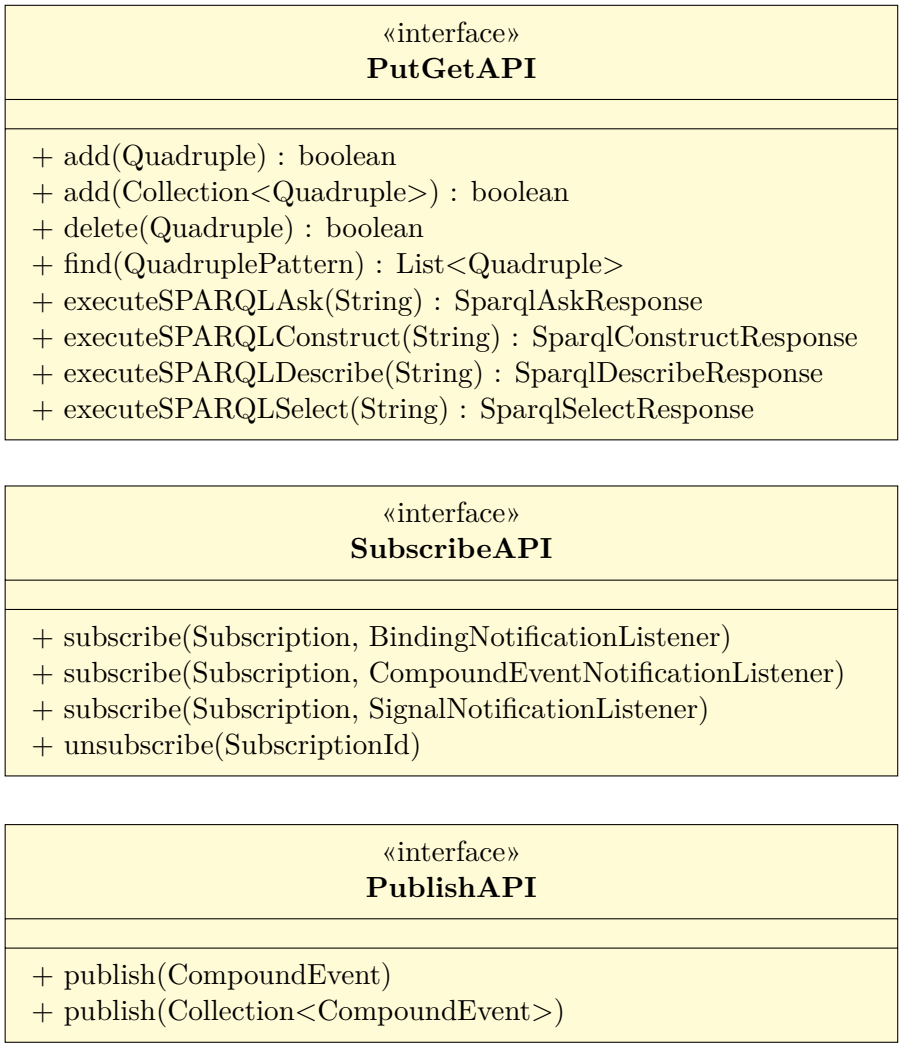


Figure 6.7 – Shortened public API with essential methods exposed by related EventCloud proxies.

in charge to create an execution graph for subqueries (atomic and range queries) extracted from the input query thanks to the second software element that is the *QueryDecomposer*. Then, the third traverses vertices (i.e. subqueries) to dispatch them in parallel (in our current implementation) with the help of a *MessageDispatcher*. Although some software elements are limited to a unique component, some others such as the *QueryDecomposer* is shared between a *PutGet Proxy* and a *Subscribe Proxy*. Indeed, the later also needs to extract sub-subscriptions which are in fact subqueries since SPARQL is used as the subscription language. Similarly, all proxies interact with a structured P2P network through message passing and thus, share the same *MessageDispatcher* software element provided from the proxy abstraction introduced in Section 6.1.1.

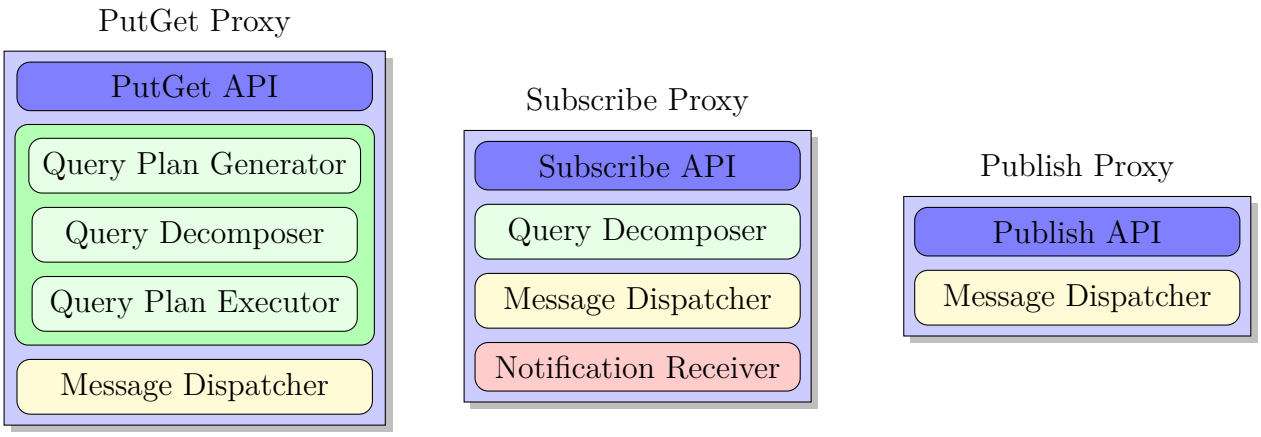


Figure 6.8 – Internal EventCloud proxies architecture.

To improve interoperability, the different proxies presented above may be exposed as a SOAP web service. Furthermore, it may be decided that publish and subscribe proxies interfaces are exposed by using our own WSDL or the WS-Notification [154] one that aims to standardize publish/subscribe interactions in web services. In this last case, translators are provided by the EventCloud middleware to translate plain old XML payloads to and from XML[2].

[2] `http://goo.gl/cv3iJH`

EventCloud peers

Peers internal is much different than proxies. However, peers behave the same and, consequently, share software elements. Figure 6.9 highlights these elements whose most of them do not work in isolation. Rather, they require frequent interactions. Below, their functions and their relations are explained.

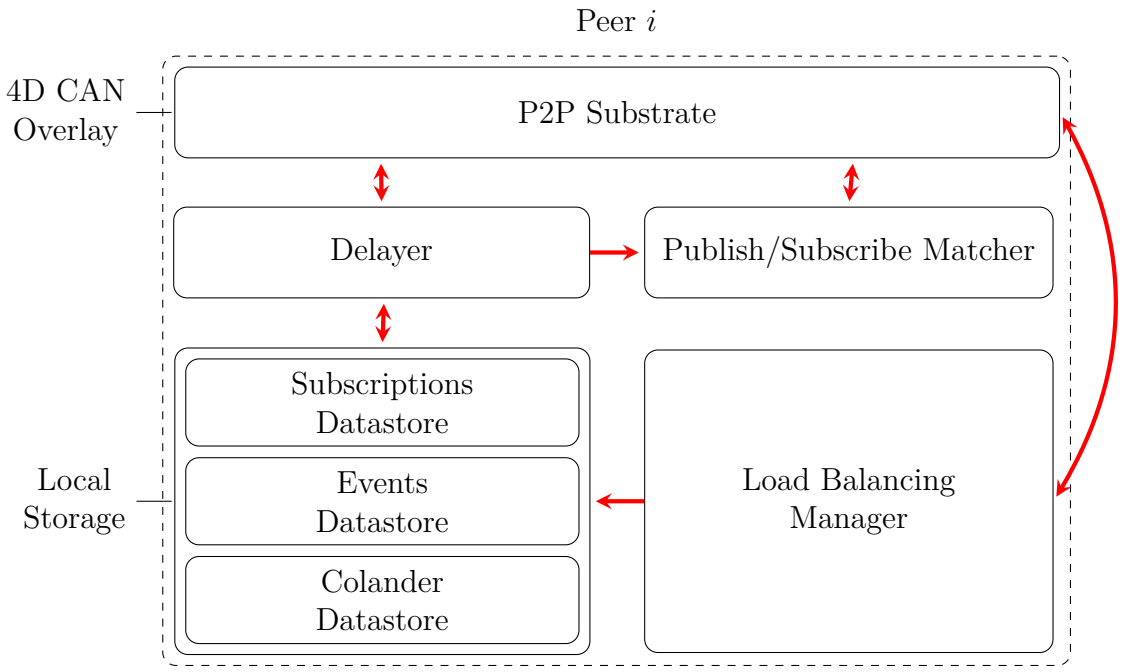


Figure 6.9 – Internal EventCloud peer architecture for an arbitrary peer i. Red arrows depict reusable abstractions.

P2P substrate This software element is responsible for maintaining the CAN infrastructure, routing messages and accessing the local storage through the delayer abstraction. The 4-dimensional CAN overlay is managed through an extended version of the CAN overlay provided with the abstract CAN library. Extended because the overlay has to interact with EventCloud specific software elements such as the semantic repositories it embeds, a publish/subscribe matcher, or even a load balancing manager it may optionally run. As a reminder, the default CAN overlay is responsible for maintaining a description of the zone managed by the current peer and an up-to-date list of its neighbors only.

Regarding routing, the algorithms implemented at the lower level require to compare peer's zone coordinates with coordinates extracted from quadruples. To make it possible, a unicode coordinate implementation is provided so that coordinate values may be floating points in radix different from 10. Manipulations on digits from coordinates to perform for instance a split during a join operation are performed with the help of the Apfloat [155] high performance arbitrary precision arithmetic library. However, comparison between a quadruple coordinate and a zone coordinate is made by comparing directly unicode values since they are real numbers represented in the same radix. Let's consider for example $zc_l = subj$ and $zc_u = uzo$, two zone coordinates, respectively the lower and upper bound coordinates a quadruple coordinate $q_c = http://example.org/subject$ has to be compared to. The first step consists in checking whether an upstream load balancing strategy is enabled. In case it is, for instance prefix removal, the associated function is applied as explained in Section 5.2.2. Thus, $q_{cs} = subject$ is obtained. Then, q_{cs} is compared to zc_l and zc_u, character by character by using at most all the characters from zc_l and zc_u, as it would be done with digits from decimal numbers. In that case, q_{cs} is detected as greater than zc_l and lower than zc_u, thus q_c is assumed managed by the peer zone on the specified coordinate dimension. The consequence of this scheme is that the extra cost of working with big radix remains acceptable since it has little impact on the routing performance.

Delayer The delayer software element acts as an intermediate between the local storage, the P2P substrate, and the publish/subscribe matcher abstraction. Its purpose is to buffer incoming data that have to be written to the disk because the commit of data to disk is an expensive operation. Also, since data require to be stored before being notified, the matching between buffered data and subscriptions is delayed. Buffered data and subscriptions are sent to the publish/subscribe matcher for checking the satisfaction between events and already stored subscriptions (and/or symmetrically between subscriptions and already stored events) when they are flushed from the buffer only. The main benefit to delay the matching is that it leads to fewer queries to perform on the local storage. Without buffering and delaying a query has to be performed per event or subscription received in order to find matching payloads whereas when delaying is enabled, all queries may

be wisely merged into one that is executed only once, when the buffer is flushed.

Internally, this software element implements a configurable policy to flush buffered data and trigger delayed operations when a threshold is reached (e.g. the time passed since last commit aka commit interval, number of elements received such as quadruples, compound events or subscriptions). The potential impact of this solution on the overall performance of the system is discussed in Section 6.2.3.

Publish/Subscribe matcher The publish/subscriber matcher is the software element in charge to apply one of the Publish/Subscribe algorithm (CSMA or OSMA) we have presented in Chapter 4. It registers subscriptions and checks the satisfaction of subscriptions with events it receives (and already stored). Additionally, the matcher caches locally the most recently used subscriptions to avoid as much as possible expensive interactions with the local storage.

Local storage The local storage is ultimately responsible for storing data and processing queries locally. It is important for the P2P infrastructure to be independent from the storage implementation. All references are isolated through an abstraction layer whose the role is to manage the differences between data structures and API between the P2P and the storage implementations. However, some requests require the access to a local datastore to read or write some information.

The storage abstraction is bound to three datastores that, each, encapsulates a Jena TDB [82] instance and optionally a stats recorder. As depicted by Figure 6.10, the stats recorder instance may be used by the load balancing manager to retrieve an estimation of the load in terms of quadruples, and this per dimension because the centroid or mean is computed per RDF term value from quadruples.

Regarding the datastores, one is used for storing subscriptions, another to store RDF data that are added synchronously or published asynchronously (events datastore), and a last to filter intermediate results returned during the execution of a SPARQL query (colander datastore). This last datastore is used mainly to compute joins between quadruple sets returned with the execution of atomic and range queries during the handling of SPARQL query. Besides it is also used to support SPARQL operators that are not managed in a distributed manner. Using three datastore instances has several benefits. In our case, they are at least two reasons

for having an independent subscription and events datastore. First, it allows to easily distinguish subscriptions from business data without having to add a flag, additional quadruples or even to create a complex quadruple representation. Second, a Jena TDB instance supports transactions with multiple concurrent readers but only one writer at a time. By using two datastore instance, concurrent writes against the subscription datastore and the events datastore can be performed at the same time.

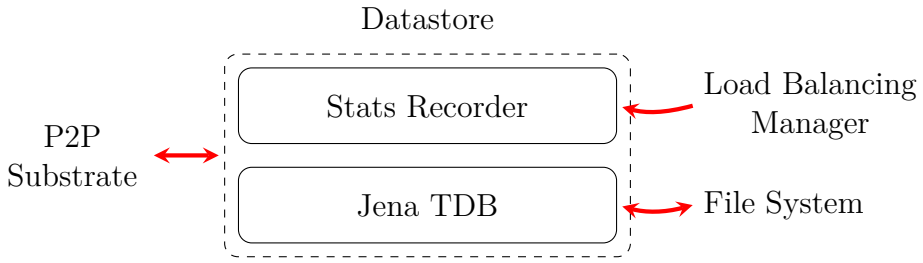


Figure 6.10 – Internal architecture of datastores (subscriptions, events and colander datastores) embedded by peers' local storage. Red arrows depict reusable abstractions.

Load balancing manager The load balancing manager, when enabled, encapsulates a load balancing strategy to apply in order to share load imbalances. It may rely on the load balancing strategies introduced in Section 5.2.2, namely *absolute* or *relative*. To achieve its purpose each load balancing strategy has to measure the load associated to its peer. Since our criteria are about RDF data, this is made through interactions with the local storage that maintains statistical information about data distribution per dimension. Interactions with the P2P substrate are also required when load information should be exchanged with neighbours or other members of the P2P network, especially with the relative load balancing strategy. Regarding this last, the manner load information is disseminated has its own abstraction which allows to plug different gossip strategies. Although only imbalances related to quadruples are taken into consideration for the moment, load balancing software elements has been design to support multiple independent criteria. Finally, how imbalances are fixed with the help of peer allocations or relocations is isolated in a dedicated software element that enables sharing between criteria.

6.2 Performance Tuning

Throughout the implementation of the EventCloud middleware, we identified different bottlenecks. The purpose of this section is to summarize different solutions we have proposed, implemented and evaluated to enhance the performances of the whole system.

6.2.1 Multi-active objects

Initially, the middleware has been implemented by using standard Active Objects with Immediate Services (ISs). ISs enable remote method invocations to be processed immediately by spawning new threads. The benefit is that they allow to take advantage of multicore machines whereas the default serving policy provided by ProActive, which enqueues incoming requests and handle them one by one through a single thread, cannot. However, the main drawback is that ISs break Active Objects' semantic. Therefore, all ISs calls are handled synchronously which entails the caller thread to block until it receives the results. Besides, ISs do not impose a limit about the number of threads to spawn and require to manage mutual exclusion manually, which allows finer control than MAOs but at the price of more complex and longer pieces of code to write.

Once a first version of the MAO library was available, we started to analyze its use inside the EventCloud middleware. The first step was to assess and compare the performance between an implementation of our solution with ISs and another with multi-active objects [156]. As a result we observed that the solution based on MAOs is in the best case identical in terms of performance obtained or even slower to our previous implementation based on ISs. It suggests the original implementation was already well parallelized. However, MAOs require less threads to achieve almost the same performance. The reason lies in the fact that, unlike ISs, MAOs do not have to use a dedicated thread to perform an asynchronous remote method invocation since methods without return type that are not declaring to throw an exception are handled asynchronously.

Thereafter, the MAO implementation has become more stable and we decided to keep it for the EventCloud middleware and to investigate how performances

could be tweaked. This lead to an empirical definition of soft and hard limits used with AOs and the introduction of priorities, as explained below.

Hard and soft limit definition

Defining properly MAO limits is a tricky process because it implies to consider many factors. Indeed, value must be set according to the machine architecture where experiments are made but it also depends of requests' type executed, their frequency, the compatibilities defined and the type of task performed once a request is executed. For instance, requests performing I/Os will most likely slow down the execution of other requests that are a consequence of the one being executed. Increasing the number of threads that may be executed concurrently may be advantageous if some independent and non I/O intensive requests (i.e. memory bound) can be processed while I/O requests are in progress, otherwise the benefit is less obvious. Another non trivial issue to monitor that may inhibit the effect of increasing MAO limits relates to threads contention. The situation occurs when multiple threads are waiting for a lock that is currently being held by another thread. In that case, boosting the number of threads heighten contention and makes the application exhibit worse performance.

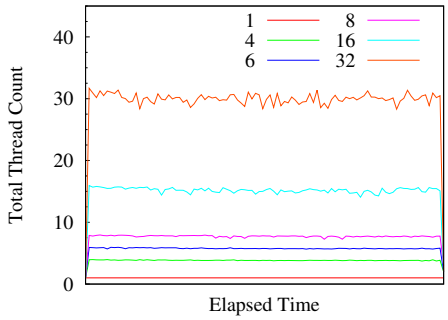
In the following we describe how MAO limits have been empirically defined for the publish/subscribe layer. In this context, entities that are involved are publish and subscribe proxies but also peers. Publish proxies, which do not trigger reentrant calls, define a hard limit that prevents to use more than t threads at any moment for handling requests. Publishing events is an asynchronous process that is really fast, experiments showed us that increasing t does not increase throughput at the subscriber. Peers simply enqueue requests faster but since they perform routing and matching at a slower rate than the one at which publications are received, increasing t is not beneficial. For this reason we set t to the number of cores available on the machine where the publish proxy is running.

The next step consists in finding the best MAO limit for peers. Peers define a soft limit that restricts the number of threads running but allows, as opposed to a hard limit, an unlimited number of threads to wait (e.g. for futures) in order to prevent deadlocks with reentrant calls. To help us to choose properly the

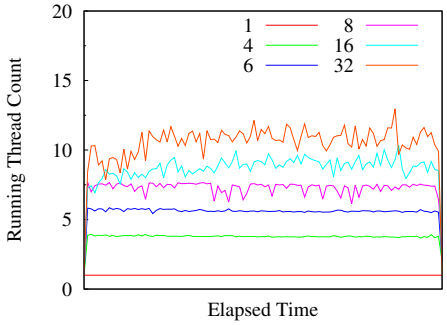
appropriate limit value, we have performed experiments based on a configuration with OSMA involving one publish proxy, one subscribe proxy and one peer, each on a dedicated machine with 8 cores. By pushing the handling capacity of the peer to its limits and deciding MAO limit based on this scenario, the value remains also suitable for configurations with multiple peers. To this aims, 15000 CEs are published, each embedding 5 quadruples. Only one path query subscription with $k = 5$ (the number of quadruple patterns linked together) is registered.

Intuitively, to assess MAO limit at a low level, a method is to trace the number of threads used by the MAO framework during a benchmark execution for different soft limit values. We performed this assessment in a first experiment and we have got the results depicted on Figure 6.11(a). These last clearly show that at any time the number of threads used is almost equals to the value set for the soft limit, thus suggesting the peer is running at its maximum capacity. However, by considering all threads without taking into account their state (i.e. whether they are running or waiting), nothing more may be deduced.

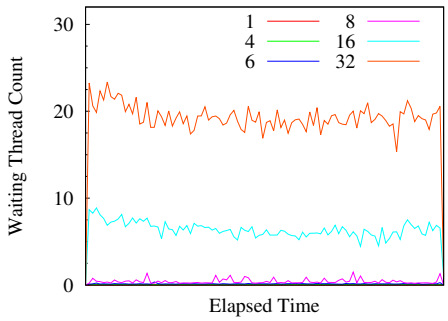
Consequently, our next two experiments have consisted in varying, again, the soft limit from 1 to 32 and, for the values used, to trace the number of threads running and waiting. In our case waiting threads are threads which are blocked for a lock acquisition in order to enter a synchronized block or method at the EventCloud level since no future is triggered given that all publish/subscribe method calls are asynchronous with OSMA. Results are depicted respectively on Figure 6.11(b) for running threads and Figure 6.11(c) for waiting threads. The former figure shows that no more than 13 threads are running at any time whatever the soft limit is. Thus, setting a value higher than 13 for the soft limit has no benefit. The latter figure allows us to approximate the optimal value we are looking for but also to identify a potential bottleneck. Indeed, after a thorough check, all waiting threads are in fact blocked threads waiting for acquiring the lock on our delayer datastructure that acts as a cache layer for handling new events. Since the number of threads waiting is rising when the soft limit is greater or equals to 8, it strongly suggests contention among threads to obtain the lock to our delayer datastructure. A solution could be to explore a new implementation that entails less locking to improve scalability. However, this is left as a future task since many optimizations we have made may no longer hold, thus requiring intensive tests and benchmarks.



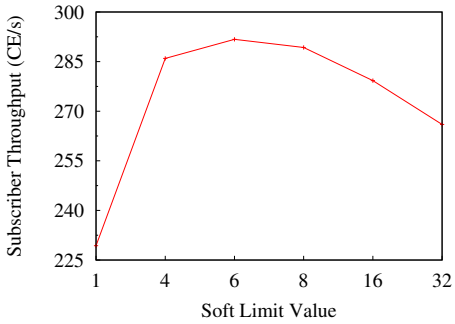
(a) Threads usage by varying soft limit value on peer.



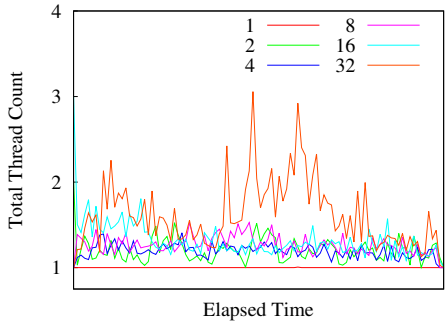
(b) Running threads usage by varying soft limit value on peer.



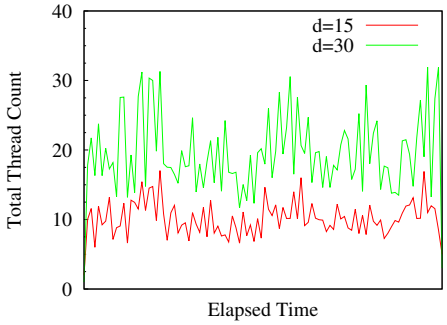
(c) Waiting threads usage by varying soft limit value on peer.



(d) Subscriber throughput when varying soft limit value on peer.



(e) Threads usage on subscribe proxy by varying number of peers.



(f) Threads usage on subscribe proxy with 16 peers by simulating delivery actions.

Figure 6.11 – MAO soft limit evaluation for peers and subscribe proxies (threads usage is approximated with a bezier curve to get readable figures).

In conclusion, it seems more appropriate to set the soft limit to 6 than 8 for peers deployed on 8 cores machines since this value incurs less contention among threads. To confirm this choice, we have performed a new experiment that compares the subscriber throughput obtained for different MAO limits. Results depict on Figure 6.11(d) show that the throughput increases when the soft limit varies from 1 to 6 but declines for higher values, hence strengthening our previous finding. For information, this last experiment has not been presented earlier because it may be argued that results are affected by some other layers of the application since subscriber throughput is measured at the subscriber side which is outside the P2P network. Previous experiments eliminate this eventuality.

Finally, we have studied subscribe proxies. A subscribe proxy may receive notifications from multiple different peers at the same time. Furthermore, subscribe proxies' requests are much less exposed to contention than peers' requests because a subscribe proxy makes use internally of concurrent maps for marking an event as received but also for finding to which subscription a notification is intended for. Based on these two observations, the next experiment adopts the same benchmark configuration as before but aims to compare the total number of threads used on the subscriber when the number of peers involved is increased from 1 to 32 by power of two for a soft limit set to 32. As depicted by Figure 6.11(e), the number of threads increases to some extent with the number of peers but never exceeds 3. The reason lies to the fact that a subscribe proxy handles notifications quickly but at a rate that is bounded and slightly slower than the one at which they are sent by peers. To corroborate our discovery and to show that threads usage depends also in part of the delay at which notifications are handled at the subscriber side, we performed an experiment that varies the delivery time. In this purpose, the listener registered with the initial subscription on the subscriber makes sleep for d milliseconds the thread delivering the notification. This way, the listener is simulating an action taking approximatively d milliseconds to handle each event notification. Figure 6.11(f) depicts results for two runs, one with d set to 15 and another with d set to 30. Compared to results on Figure 6.11(e) where d was set to 0, we observe that threads usage is greatly increased, thus suggesting that delivery actions taking time may take advantage from a high MAO limit. However, this limit must be selected carefully, especially if the action involves locking since

this last generates contention among threads thus causing poor performance when MAO limit is increased, as we have seen for peers.

To summarize, selecting the appropriate MAO soft limit for subscribe proxies strongly depends of the action performed with the listener registered along with the subscription, which is use case dependent. However, since no contention occurs by default and because threads are spawned only when required and garbage collected a few time after they complete, we set the default soft limit value equals to the number of cores available on the machine where the subscribe proxy is deployed to face potential overload if the number of peers is sharply increased.

Priorities

Multi-active objects enable concurrent requests execution but the default serving policy does not allow to control scheduling between compatible requests. This lack of control may lead to poor response time. Let's consider a concrete example we have faced with while experimenting the first version of our publish/subscribe algorithm based on polling. In this scenario, events are emitted to peers in batch through request of type P. Then, peers perform the matching and forward a notification to subscribers that trigger a reconstruction based on an identifier. The reconstruction consists in routing one or more synchronous requests of type R for retrieving chunks that make up the event a subscriber has been notified about. Once requests of type R reach peers, they are queued into the request queue of their respective peer. However, requests of type R that are the last actions to complete before delivering events are not served until requests of type P (which arrived before in the request queue) are executed or the last requests of type P that predate requests of type R are being executed and some threads become free, despite the fact that requests of type R are compatible with requests of type P. Requests are simply scheduled in the order they are detected as compatible. This last depending of the order in which requests are queued in the request queue. Consequently, in the presented example reconstruction requests are delayed, thus having a negative outcome on the latency perceived by subscribers to receive matching events.

In summary, our goal was to propose a mean to improve efficiency by reducing the scheduling time of critical requests. To this aim, we have extended the

multi-active framework and its meta language based on annotations to enable the definition of priorities. Priority relationship between requests is based on integers. Listing 6.2 gives an example for improving the scheduling time of reconstruction requests discussed above. The set of priorities is embedded by a `@DefinePriorities` annotation at the header of an AO class. Inside, priorities are defined with the help of the `@Priority` annotation which requires at least the method `name` and the priority `level` attributes, others are optional.

```
1 @DefinePriorities({
2     @Priority(level=1, boostThreads=1, name="route",
3               parameters={ReconstructCompoundEventRequest.class})
4 })
5 public class SemanticPeerImpl {
6     // ...
7 }
```

Listing 6.2 – Priorities definition.

Priority level is an integer that ranges from -2^{31} to $2^{31}-1$. Methods that satisfy no priority constraint are assigned to a default and implicit priority constraint with priority level 0. The method name defines method calls that are assigned to the priority rule. The optional parameter attributes are used to assign the priority constraint to method calls whose parameters are of the specified types. The `boostThreads` attribute is useful to prevent starvation issues that may occur when priorities are applied. It defines the number of extra threads that may be used by the MAO framework in addition to the number of threads permit for the hard or soft limit. However, boost threads are spawned when the soft or hard limit is reached only. Finally, priorities are also settable programmatically to have the possibility to enable or disable priority rules according to properties.

Internally priorities are implemented with a few changes to the request executor provided with the MAO framework. By default incoming requests are put in a request queue, then compatible requests are pulled out in a ready queue. Once a thread completes a request execution, a new request is picked from the ready queue for execution. Our priority selection is applied during this last phase. Thus, a request may overtake another request according to priorities defined if both

requests are compatible.

To assess the effect of our solution we executed the scenario described previously with and without priorities. Results are sketched on Figure 6.12. Enabling priorities allows to keep reconstructions time almost constant whereas without priorities the reconstruction time depends of the position at which reconstruction requests have been enqueued compared to publications. Therefore, when a burst of publications predates reconstruction requests, these last take up to 10000 times longer to complete without priorities than with priorities.

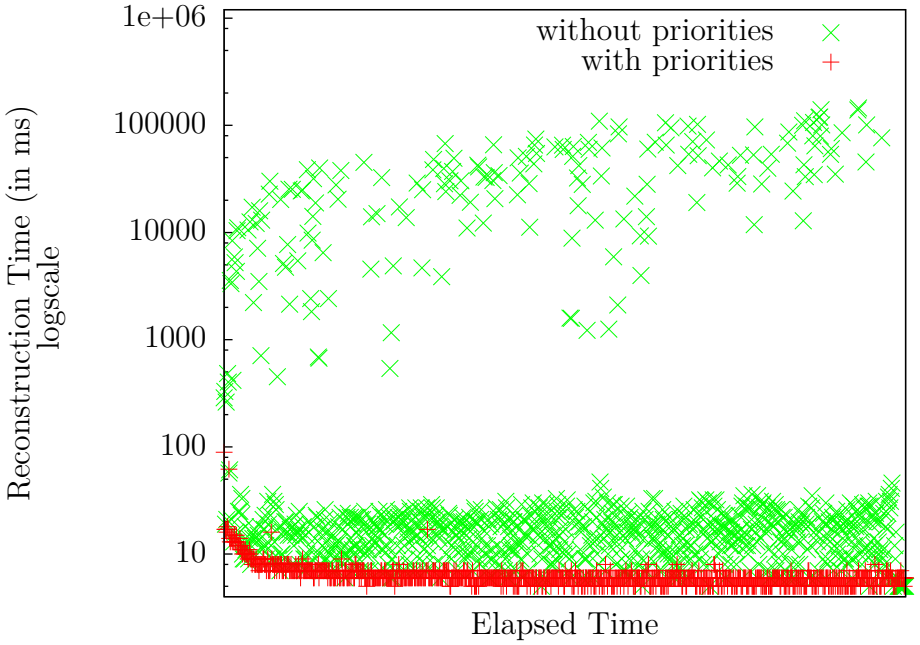


Figure 6.12 – Priorities effect on reconstructions with CSMA.

The presented solution has been implemented and integrated in the ProActive code base. Now, Justine Rochas is investigating within the context of her thesis a new manner to express priority relationship based on a dependency graph [157].

6.2.2 Serialization

The mechanism provided by default with Java to marshall objects, which simply consists in implementing a *Serializable* interface, is really handy but not efficient,

especially for exchanging messages in a publish/subscribe system where responsiveness plays an important role. The main issues are the following:

- Marshalling and unmarshalling require the serialization mechanism to discover information about the object instance it is serializing. Using the default one, all field values are discovered by reflection;
- Classes that do not define a *serialVersionUID* field, that is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible, compel the default serialization mechanism to compute one. This involves going through all fields and methods to generate a hash value;
- Serialized classes and their serializable superclasses have their descriptions sent in the output stream.

To solve the performance problems mentioned above, Java allows to define our own serialization behavior through the *Externalizable* interface. Obviously, this efficiency comes at a price, that of defining manually how fields are encoded and updating this definition when the characterization of the class to marshall changes. In our case, we have used externalization (i.e. custom serialization) for our custom RDF data types wrapping existing ones from the Jena API. Since data types provided by Jena are not serializable, the overhead induced to support custom marshalling was not too high in comparison of using the default mechanism. The benefit of using custom serialization or frozen objects, as we will explain below, is not to save bytes to exchange but mainly some time required to encode and decode exchanged information.

Frozen objects

Whether it is to update the state of some peers, to handle SPARQL queries or more generally to execute a request, messages are exchanged between peers. By default, a deep copy of the message that is sent on the wire is made for each remote method call that routes a message to its destination. Since messages are routed from peers to peers in multiple steps, the data that are piggybacked by messages

are serialized and deserialized at each hop whereas only the final receiver is most of the time interested by the information that are conveyed within a message. To prevent expensive serializations, we have introduced the concept of frozen objects that aim to fully serialize and deserialize transferred objects only once.

To freeze an object graph, we propose a simple *FrozenObjectGraph* class that encapsulates an object graph which has been previously serialized as a byte array. This class has also the particularity to implement the Java *Externalizable* interface in order to define its own serialization behavior and thus avoiding the overhead from the default Java mechanism. In addition, a *FrozenObjectGraph* disposes of one method called *unfreeze* that deserializes the original object from the byte array and, thus, returns a deep copy of the initial object that was frozen. This last method is used to retrieve back the object value, once the message that embeds the frozen object is required. It is usually invoked when the message reaches its final destination.

Compared to a scenario where a message routed in N hops requires N full serializations and deserializations, up to $N-1$ expensive operations may be saved when frozen objects are used. Indeed, with frozen objects, the value is fully serialized and deserialized respectively before the first hop and after the last hop. Consequently, only 1 full serialization is required. The remaining $N-1$ serializations are made on byte arrays, which is more efficient than a full object serialization on non primitive data types.

To assess the effect of using *FrozenObjectGraph* objects, we have performed measurements by varying two parameters: the type of the object to freeze and the size of the object that is frozen. The results show the time required to perform 4×10^4 deep copies (serialization + deserialization) of a message with and without the use of frozen objects for a message that embeds either a simple quadruple or a compound event made of 10 quadruples. The number of serializations we propose to use corresponds to approximatively 10^4 messages to route if we assume that messages are routed on a 4 dimensional CAN network with 256 peers since the average routing path in this configuration is equals to 4. Results are depicted on Figure 6.13. They show the percentage difference, in terms of time required between the solution that relies on the default serialization mechanism provided by Java and the one that uses frozen objects, when the message that is deeply

copied embeds either a quadruple or a compound event. When a quadruple is used, we can see that the usage of frozen objects is interesting when the sum of characters contained by RDF terms is greater than 128. In this last case, the use of frozen objects improves the execution time by 3% and up to 27% when the size of a quadruple reaches 512 characters. The effect of using frozen objects is strengthened when an object with a larger object graph, such as a compound event, is considered.

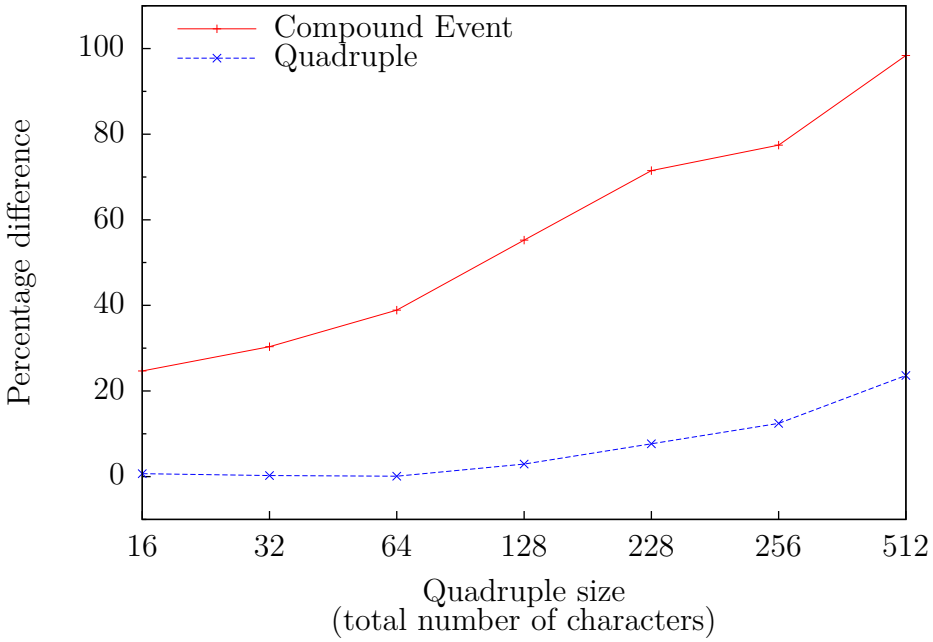


Figure 6.13 – Message serialization with and without the use of frozen objects.

Quadruples' size is really scenario dependent, however to evaluate the benefits in a standard and realistic use case, we have measured the average size of quadruples for 10^8 entries from the DBpedia dataset provided for the Billion Triples Challenge 2012[3]. As a result we get that quadruples contain in average 228 characters. For this size, frozen objects improve the marshalling time by 8% with a simple quadruple and by 71% when a compound event with 10 quadruples is used. The effect of using frozen objects could be further improved by using compression but it is let for future work.

[3] `http://km.aifb.kit.edu/projects/btc-2012/`

6.2.3 Local storage

As we explained previously, the local storage on peers is made of multiple datastore instances, each one being bound to a Jena TDB repository. The main reason is that it reduces contention when concurrent read/write operations must be performed for events and subscriptions at the same time. To better improve the time required to execute SPARQL queries, we have introduced some rules applicable for queries involved in the publish/subscribe matching. First, SPARQL queries are not passed to the Jena engine as simple strings. Instead, SPARQL queries are manually built as a SPARQL Abstract Syntax Tree (AST)[4], also known as an algebra in the Jena terminology. It prevents parsing a sequence of characters and performing a lexical analysis for queries that are always built using the same model. Second, static optimizations applied by Jena on queries (which consist of transforming a SPARQL algebra before its query execution begins by reordering filter expressions, BGP, etc.) are disabled. It is assumed that queries we build are already optimized.

Subscriptions storage How subscriptions are stored and detected as matching events requires some explanations. To keep persisted information homogeneous it has been decided to store subscriptions in RDF. Doing so entails to define how subscriptions are expressed in RDF. Listing 6.3 gives an example for the SPARQL subscription presented in Listing 4.1. The first eight quadruples (line 1–9) are meta information about the subscription, namely its identifiers, the content of the full query, its type that defines how solutions have to be notified (i.e. as bindings, CEs or signals), the time at which it enters the P2P network, and a subscriber reference to send back matching events. Multiple identifiers are associated to a subscription because, as explained in Chapter 4, a subscription may be rewritten several times. Initially, the original subscription identifier (oid) and the current subscription identifier (id) are set to the same value but the parent identifier (pid) is undefined. Once a rewriting is performed, for instance S is rewritten to S', S'_{pid} is set from S_{id} and S'_{id} to a new value that uniquely identifies the rewritten subscription. Chaining identifiers is useful for unsubscribing as explained in 4.2.3, but also to detect some situations. For instance, let's take as an example a subscription S received on a

[4]An AST is a tree representation of the abstract syntactic structure of a language or grammar associated to a language where leaves depict tokens of the grammar.

peer for indexation. If an event is detected as matching S, its indexation time lower than the one for the matching event and S_{pid} is undefined, then its means that an ordering issue occurred for a just submitted subscription.

To provide fast access to quadruples related to a specific subscription, its metadata or subsubscriptions' metadata, the graph value of each quadruple from a RDF subscription payload contains an URN identifying the subscription (e.g. like *urn:ec:s:1* on Listing 6.3). Similarly, the subject value embeds either the original subscription identifier or a subsubscription identifier (e.g. *urn:ec:ss:1*). In the former case it allows fast access to subscription metadata whereas in the latter it grants direct access to subsubscription metadata.

Finally, the most important decision is how subsubscriptions are modeled to detect if subscriptions are fulfilled for a given CE or quadruple. A simple solution is to store the SPARQL query as String and when required, to load back all subscriptions, inserting the CE or the quadruples to check the satisfaction with into an in-memory dataset and to execute each subscription against the dataset to find back solutions. Although the implementation is easy, it does not scale with respect to the number of subscriptions. To address the issue we propose to split each subsubscription from a subscription into pieces, as depicted on lines 10–17 and 19–26. Fixed RDF terms values are put as object value of quadruples. When a variable is declared in a subsubscription, a custom URN is used instead. This way, it is possible to define a matching SPARQL query that finds identifiers of subscriptions that are satisfied by a given CE or quadruple. In an abstract manner, subscriptions are transformed as data and events as simple query patterns. It may also be noticed that each subsubscription defines a quadruple containing as object value the name of the variables involved in the subsubscription (see lines 17 and 26). The purpose is to help solving subscriptions registered with a binding listener since in that case values associated to variables only must be notified.

Unfortunately, subsubscriptions involving filter constraints (range queries) are not addressed by our previous discussion. Currently, to support range queries we apply the solution we presented without considering filters and then, thanks to an extra quadruple added to the RDF representation that defines if the original subscription contains filters, we are able to detect if solutions found require refinements. In this last case, intermediate solutions are put in an in-memory dataset

and refined by reexecuting the original subscription. Ideally, the expression tree associated to filter constraints should be tokenized and put in way that allows the matching SPARQL query to consider filter constraints.

```
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:oid, "1")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:pid, "1")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:id, "1")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:query, "SELECT ...}")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:type, "2")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:itime, "2013-12-12T11:01:14.977")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:subscriber,
            "rmi://oops.inria.fr:1099/subscribe-proxy-40c1d51b")
(urn:ec:s:1, urn:ec:s:1, urn:ec:s:iref, urn:ec:ss:1)
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:id, urn:ec:ss:1)
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:index, "0")
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:g, urn:ec:var)
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:s, urn:ec:var)
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:p,
                                    http://example.org/v/action)
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:o, "exits")
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:varnames, "g=g,s=id")
...
(urn:ec:s:1, urn:ec:ss:1, urn:ec:ss:id, urn:ec:ss:3)
(urn:ec:s:1, urn:ec:ss:3, urn:ec:ss:index, "0")
(urn:ec:s:1, urn:ec:ss:3, urn:ec:ss:g, urn:ec:var)
(urn:ec:s:1, urn:ec:ss:3, urn:ec:ss:s, urn:ec:var)
(urn:ec:s:1, urn:ec:ss:3, urn:ec:ss:p,
                                  http://xmlns.com/foaf/0.1/age)
(urn:ec:s:1, urn:ec:ss:3, urn:ec:ss:o, "exits")
(urn:ec:s:1, urn:ec:ss:3, urn:ec:ss:varnames, "g=g,s=id,o=age")
```

Listing 6.3 – SPARQL subscription representation in RDF.

Buffering and delaying

Jena TDB provides support for transactions. In our implementation transactions are used to avoid unexpected or undesirable results when interleaving operations on a datastore (read/write) occur due to the execution of some requests in par-

allel. Transactions ensure ACID properties: atomicity, consistency, isolation and durability. Although the last is crucial to ensure that information persist through crashes, not all scenarios require this level of confidence. Moreover, ensuring no information loss would require much more work, especially at the P2P level by replicating events and subscriptions but also at the ProActive level by storing incoming requests in a safe manner to replay them in the same order. Since fault tolerance is not the main contribution of this thesis, we assume some loss of information is acceptable. Nevertheless, the manner the publish/subscribe algorithm is designed implies that events which are not stored are not notified, therefore subscribers cannot reference events that are potentially lost. To prevent loss against power outages or hardware component failures, buffered data should be journaled, with the overhead it induces.

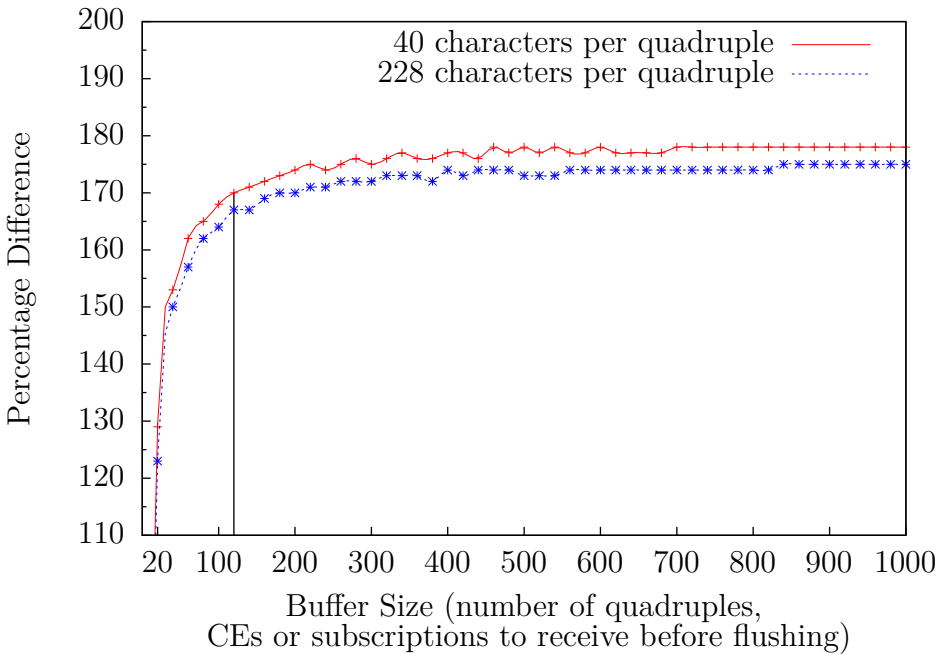


Figure 6.14 – Delayer benefits when varying buffer size.

Even if the advantage of buffering data and delaying operations is rather straightforward, experiments are required to determine the buffer size under which the best throughput is obtained, in particular with OSMA, when data and operations are handled in batches. To achieve this empirical assessment, the benchmark

has been set up on a single machine (with a SAMSUNG HD103SJ hard disk drive) hosting one peer, a publisher publishing 5000 CEs and a subscriber that registers a path query. Each CE contains 5 quadruples but the size of each quadruple remains an experiment parameter. The policy applied by the delayer is to flush data when the buffer is full (its size in terms of quadruples, CEs or subscriptions is reached) or $500ms$ have elapsed since the last commit. In a first experiment, we vary the buffer size from 20 to 1000 for two different size of quadruples: 40 and 228 characters. To compare results, the percentage difference in terms of throughput (CE/s) at the subscriber side is computed. The calculation is performed between values obtained for the buffer size tested and the one got for a buffer of size 1 which simulates a disabled delayer. Figure 6.14 clearly shows the benefit of using a delayer. The throughput is increased by up to 179% and 176% when respectively 40 and 228 characters are used per quadruple. It may also be noticed that both curves follow the same trend even if the size in terms of characters per quadruple differs. The throughput, and thus the percentage difference skyrockets as soon as the buffer size increases but quickly stabilizes. Since less than 5% improvement is achieved when the buffer size exceeds 120, it has been decided to select this last value as the default one for the delayer policy.

Although both curves follow the same trend when CE size and consequently the buffer size is low, increasing greatly CEs size decreases performances earlier. To highlight the issue but also to identify possible reasons, we replayed the previous experiment for larger quadruple sizes and measured, in addition to the percentage difference in terms of throughput, the average time required to flush the buffer to the disk when its capacity is set to 120.

Figure 6.15 shows the results we get for quadruple sizes ranging from 40 to 6400 characters which corresponds approximatively to 0.5 KB and 64 KB per CE. To better highlight the cause, we put in correlation on the figure the average time required to flush data to the disk. When quadruple size per CE is lower than 600 characters the commit time remains almost stable. However, when it exceeds 600, which corresponds to roughly 80 KB per CE or 970 KB by considering the buffer capacity, the time required to write buffered data to disk starts to increase exponentially. The issue is strongly visible when quadruples contain 6400 characters, which requires to have approximatively 7.7 MB in the buffer before

flushing data to the disk. Thus, we concluded the performance penalty is probably coming from Jena TDB that does not support large batch commits. When large payloads have to be considered, a solution could be to monitor incoming CE size and to adjust the buffer size accordingly.

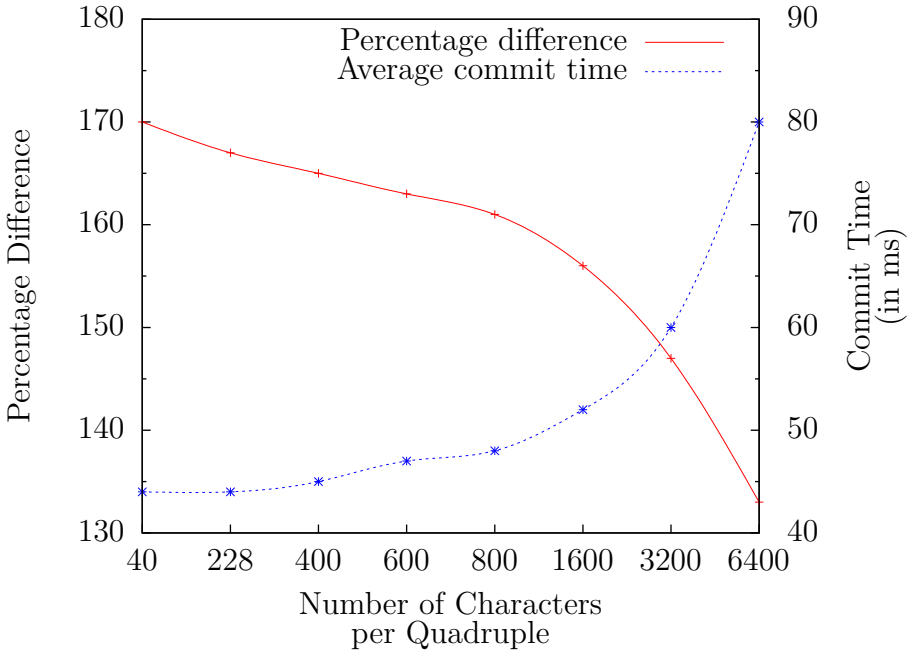


Figure 6.15 – Quadruple length effect on buffering.

Summary

To validate and evaluate our solution along with the proposed algorithms, we have designed and implemented the EventCloud middleware which is a computer software that provides a distributed datastore service on top of a 4-dimensional CAN network to store and retrieve quadruples through SPARQL but also to manage events represented in RDF. The EventCloud middleware has been implemented in Java. It relies on the ProActive Programming library and its multi-active objects extension to respectively distribute components and leverage multi-core processors by handling when possible some requests in parallel to others. Special attention has been given to make the architecture modular, thus improving reusability and

flexibility. Also, performance bottlenecks have been studied and some solutions proposed.

Another assessment of our solution is its integration in the platforms developed within the context of the PLAY and SocEDA projects whose the objectives and their respective architecture are presented in appendix A and B. Criteria evaluated have been performances, stability and usability [158, 159]. It is worth mentioning that some features have been encouraged by the projects we were involved in. However, their utility is not restrained to their specific contexts. It includes for instance the efforts we spent to provide XML/RDF translators or to make the deployment of the EventCloud P2P network agnostics to cloud platforms by reusing and extending abstractions provided by GCM deployment and ProActive. Another example is the EventCloud management Web service we have introduced to manage multiple EventCloud instances and its associated webapp, prototyped with Flask[5], that aims to make the management more convenient to administrators. This webapp allows to create, delete, list EventCloud instances but also proxies through SOAP messages sent to the management web service. To achieve its purpose, this last relies on multiple software abstractions such as an EventClouds registry to maintain a reference about running instances and their associated deployment configurations but also a node provider to abstract how and from where nodes from real infrastructures, going to host peers, are retrieved. Obviously, these deployment and management tools, can be reused to manage EventCloud instances inside an organization or between several. More details about the additional features built around the core of the EventCloud middleware are described in projects' deliverables [9, 160, 161, 162, 163].

The code base that makes up the core of the EventCloud contains approximatively forty five thousand lines of code along with thirty thousand lines of comments and more than 260 unit tests. The whole is distributed among six hundred and fifty files publicly available at `http://eventcloud.inria.fr`.

[5] `http://flask.pocoo.org`

Chapter 7

Conclusion

Contents

7.1 Summary

RDF has become a relevant data model for describing and modeling information on the Web while remaining fairly simple and intuitive. However, to manage and store RDF information in a continuous or batch manner has raised many questions and faced us with scalability issues when processing it in a distributed context. The main outcome of this thesis is a middleware devoted to storing, retrieving synchronously but also disseminating selectively and asynchronously RDF data.

Our first contribution relates to the design of a distributed infrastructure for storing and processing RDF data and SPARQL queries in a synchronous context by using the traditional query/response model. The architecture is based on a four-dimensional CAN overlay where RDF tuples are indexed according to the lexicographic order of their elements. This scheme avoids to use hash functions

and prevents to store the same information multiple times. Furthermore, lexicographic data indexing enables efficient support for range queries. Although basic operations like adding RDF data suffer from an overhead compared to a centralized solution, the distributed nature of the infrastructure allows concurrent accesses. Therefore, SPARQL queries are evaluated by decomposing them into subqueries that are executed in parallel.

The second contribution of this thesis is about a publish/subscribe layer for storing and selectively disseminating RDF events. We have proposed a data and subscription model respectively based on an extension of RDF and a subset of SPARQL. Furthermore, we have designed two publish/subscribe algorithms, namely CSMA and OSMA that aim different requirements. The first, CSMA, inspired by Liarou *et al.*, performs the matching with events and subscriptions sequentially but is able to fix time ordering issues between publish and subscribe operations originating from a same host. On the contrary, the second, OSMA, uses a fully distributed approach that enables to match publications and subscriptions directly in one step, thus leading to better performance but at the cost of a slightly heavier bandwidth consumption. Experiments have shown that both algorithms are complementary depending of the scenario that is considered.

Our third contribution is about load balancing. Distributed RDF systems suffer from a skewed distribution of RDF terms. Tuples with RDF terms that occur more frequently than others are indexed on a few nodes which creates hotspots. With regard to this issue, we have presented and evaluated strategies to improve RDF data balancing on our revised CAN structured P2P network. The solution we propose combines some existing techniques such online statistical information recording and gossip protocols. The former allows us to disseminate load measurements between peers so that the load balancing decision is taken by comparing peers' load with an average system load computed according to measurements exchanged. Once an imbalance is detected, previously recorded statistical information about RDF data per dimension allow us to decide how peers' zone of responsibility may be split to fairly share the imbalance. Experiments have shown that the overhead induced by online statistical data recording remains acceptable and forcing new peers to join overloaded peers by splitting the zone based on recorded information may greatly decrease load imbalances.

Finally, we have dedicated efforts to provide at the implementation level a flexible and modular middleware with clear separations between the software elements.

7.2 Perspectives

In the following we give some outlooks that can be explored with respect to the work presented in this thesis, especially some possible hints to broaden for enhancing the efficiency and the added value of the proposed middleware.

7.2.1 Optimizing query and subscriptions evaluation

As we started to highlight in their respective chapters, queries evaluation and subscriptions matching algorithms introduced could take advantage of the following research fields.

Improving query plan execution

Chapter 3 has shown that distributed SPARQL query execution depends on the complexity of the queries and the search space. Our approach consists in decomposing a query into subqueries and to execute subqueries independently and in parallel. However, when subqueries share common variables (i.e. require a join) and return tuples sets with sizes that differ from one or more orders of magnitude, our solution requires to carry from peers to peers and until the query initiator many tuples that would be unnecessary if subqueries were pipelined by previously building a sequential execution plan. Therefore, a perspective would be to provide query plans suitable for SPARQL queries involving subqueries sharing variables. Nevertheless, our parallel approach remains advantageous for independent subqueries. Thus, a step forwards would be to generate query plans by combining parallel execution of independent subqueries and pipelining of those requiring joins. Works done in [111] and [112] to respectively improving query plan execution by reordering triple patterns and estimating selectivity are probably great resources to consult. To further improve data transfers compression techniques could be applied [164]. One more consideration for improving execution time could be to remodel our overlay structure with locality awareness such that peers which are

neighbors in the overlay network are put close physically. Ali *et al.* have shown in [113] that a structured P2P system improved with locality awareness and some additional shortcuts may boost performance by a factor of two in an RDF context.

Subscriptions summarization

Designing a publish/subscribe system raises many questions. Depending of the context where it is used, one main challenge to tackle is to manage many subscriptions. Since subscriptions are entries registered on peers, when subscriptions are handled independently, the larger the number, the more expensive end-to-end delivery becomes due to increased processing. Subscription summarization has been proposed to alleviate this issue [165]. The purpose is to find subsumption relationships between interests from subscriptions in order to combine them in a summarized subscription thus reducing bandwidth consumption, storage overhead and messages exchanged between peers for performing the matching against events. Multiple approaches have been proposed these last years [166, 167, 168]. Applying such a technique to our solution will further improve performances.

7.2.2 Increasing reliability and availability

Even though routing algorithms from the proposed middleware has been designed with fault tolerance in mind to prevent a full redesign in the future, managing reliability and increasing availability has been let aside. Our middleware would strongly benefit from a solution to allow a safe recovery in case of peers failure, this incurring among others to support fault tolerance at the level of the Multiactive objects framework. Regarding this last point, more details will be provided in the upcoming thesis of Justine Rochas. In addition, a good starting point to increase availability, reliability but also read performances and thus synchronous SPARQL query evaluation could be to investigate replication. Either by studying existing replication solutions such as the one proposed in the original CAN paper [107], that proposed by Meghdoot's authors [122] or investigating new ones less sensitive to churn as proposed in [169].

7.2.3 Reasoning over RDF data

RDF is a great data model for writing globally interchangeable information. However, recording semantics or meaning requires other standards from the Semantic Web stack such as RDFS and/or OWL (cf. Figure 2.3). These last allow to define vocabularies, similarly to schemas, by defining elements used in an application, their domain, their type, their relationships but also possible constraints regarding their usage. In other words, RDFS and OWL technologies provide a solid base for understanding and thus inferring potentially relevant information. This purpose is usually materialized by computing the closure of RDF graphs which consists in making all implicit information explicit by applying all RDFS/OWL rules on RDF data until no new data is derived. Interpreting RDFS and OWL vocabularies and their entailment rules in a distributed and scalable manner remains a challenge and would be an interesting perspective to exploit the real potential of RDF [170, 171, 172].

Finally, another perspective could be to explore a solution based on a slightly adapted version of the MapReduce model for loading RDF events and handling continuous queries using SPARQL [173], with the aim to compare performances with our solution. More details will be available in the forthcoming thesis of Sophie Ge Song. In yet another direction, an outlook could be to investigate the applicability of graph databases (e.g. Trinity [174] or Stardog[1]) for storing, retrieving and selectively disseminating RDF data at very large scale.

[1] `http://stardog.com`

Appendix A

PLAY Project

The aim of the PLAY project[1] is to provide an event marketplace platform which collects information in near real-time from many, heterogeneous, and distributed event sources, processes these events in a complex manner, and, after discovering something relevant, forwards such a situation (combination of events) to the parties interested in that issue. One of the specificities of the platform being that events are represented in RDF. To summarize, the PLAY event marketplace is a framework for dynamic and complex, event-driven interactions for the Web since it enables the integration of semantic sources, the efficient management of the situation of interests (described as complex event patterns), a distributed complex event processing in order to cope with the high throughput of events, a dynamic publish/subscribe mechanism in order to enable the responsiveness in highly changing environments, and a service adaptation process that reacts on the signal from the process/environment in order to change the flow of running processes. The overall architecture, as it is depicted in Figure A.1, consists of five building blocks:

- The Distributed Service Bus (DSB) that provides the SOA and EDA (Event Driven Architecture) infrastructure for components and end user services. It acts as the basis for service deployments, and processes (BPEL, BPMN), routing synchronous and asynchronous messages from services consumers to service providers. Based on the principles of the system integration paradigm of Enterprise Service Bus. The DSB is distributed by nature. In concrete

[1] `http://www.play-project.eu`

terms, the DSB is the entry point of the platform that maps events to internal components of the platform;

- The Governance component that allows users to get information about services and events. The Governance component extends a standard Service-based governance tool by adding governance mechanisms for event-based systems. Its role is to provide ways to govern services and events. It provides standards-based APIs and a graphical user interface;

- The EventCloud that provides storage and forwarding of events. The role of the Event Cloud is a unified API for manipulating events, real-time or historic. Its purpose is to store incoming events while filtering some events to notify only those that are of interest for the DCEP that will then execute more complex queries involving time window operators;

- The DCEP (Distributed Complex Event Processing) component that has the role of detecting complex events and reasoning over events by means of event patterns defined in logic rules. To detect complex events, DCEP subscribes to the EventCloud for any simple event defined in the event patterns at a given point in time. It may also, depending of the pattern, query the EventCloud to retrieve historical information for correlating them with the ones received in near real-time;

- The Platform Services component that incorporates several functional additions to the platform as a whole. The Query Dispatcher has the role of decomposing and deploying user subscriptions in pieces supported by the Event Cloud and DCEP respectively, taking into account the expressivity supported by the two target components. The Event Metadata component stores information about events, such as source descriptions, event type schemas, etc., to enable the discovery of relevant events for an event consumer and to provide data to the subscription recommender. The ESR and SAR component form the Event Subscription Recommender (ESR) and Service Adaptation Recommender (SAR). ESR will recommend subscriptions to services based on service context and event semantics from the metadata. Thus, ESR will provide assistance to services that will have the option to be

subscribed to specific events at the right time without the services having complete knowledge about the supply in the marketplace at a given time. The objective of SAR is to suggest service administrators, changes (adaptations) of their services' configurations, composition or workflows, in order to overcome problems or achieve higher performance. Based on recognized situations, SAR will be able to define adaptation pointcuts (points in a service flow that need to be adapted as a reaction to a certain situation) and advices (what to adapt and how based on a number of service adaptation strategies).

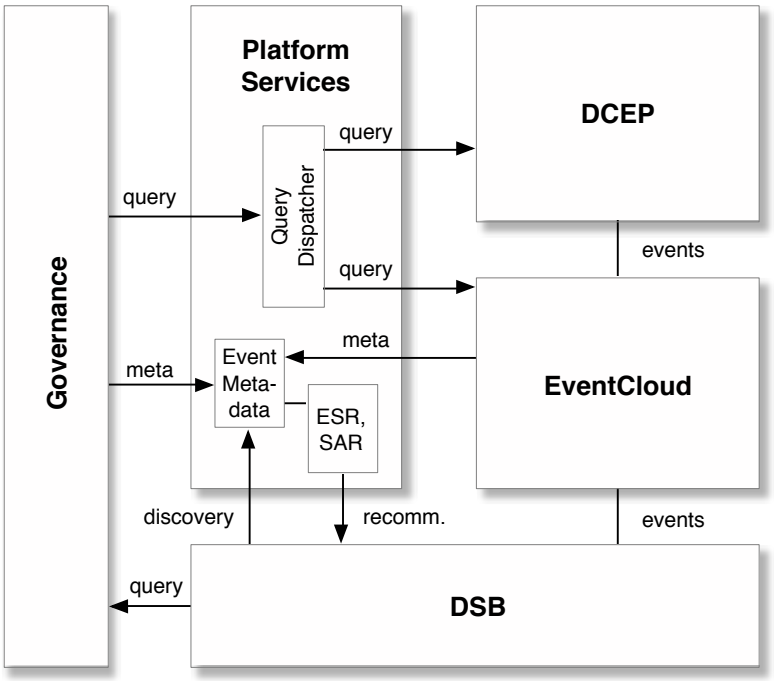


Figure A.1 – Conceptual PLAY architecture.

Appendix B

SocEDA Project

Although some aspects are similar to PLAY, the purpose of the SocEDA project[1] is rather different. Its objective is to develop and validate an elastic and reliable federated SOA architecture for dynamic and complex event-driven interaction in large highly distributed and heterogeneous service systems. Such architecture will enable exchange of contextual information between heterogeneous services, providing the possibilities to optimize and personalize the execution of them according to social network information while addressing Quality of Service (QoS) requirements. Events which are exchanged are assumed to described in plain old XML payloads to provide a platform that is compatible with legacy applications. The platform architecture is depicted in Figure B.1 and consists of the following components:

- The SeaCloud, for Services/Events Administration Cloud, is the frontal of SocEDA runtime framework. It allows clients to subscribe to a specific producer, to add CEP rules or deploy BPEL process through dedicated editors developed within the context of the project;

- The Distributed Service Bus (DSB) is an extension of the open source Entreprise Service Bus provided by PetalsLink called Petals ESB. It is made of three essential components: an adaptation service, a proxy event manager and a workflow engine. The adaptation service is designed to provide agility (seen as the combination of detection and adaptation). It allows on one

[1] `https://www.soceda.org`

hand to detect if on going processes meet the requirements of the current situation and adapt them if required. The proxy event manager maintains the list of event producers for a given topic. It can be considered as a broker for event producer. Finally, the workflow engine exposes an API to observe and command an instance of process at runtime to adapt it according to some events. This component relies on EasyBPEL, a reflexive BPEL 2.0 Engine. To summarize, the DSB enables legacy services to connect to the platform. Published events are forwarded to the SeaCloud. Services that have subscribed may either be adapted according to the situations that are induced or receive inferred facts as notifications;

- The Governance allows a user to discover all topics (type of event) and all event producers known on a service infrastructure. Additionally, it supports Quality of Services (QoS) as the definition of SLA contracts by using the WS-Agreement standard;

- The EventCloud is in charge to store all incoming events the platform receives. Moreover, it filters simple events to reduce the input of the DiCEPE component that has to perform complex correlations. Historical events are accessible by the DiCEPE to correlate historic and real-time events. Since the platform deals with plain old XML payloads, the EventCloud embeds translators to convert payload from and to RDF. Although publish/subscribe systems have the habit to hide publishers' source from the rest of the system, the EventCloud optionally allows to keep track of any publisher endpoint address in order to interact, when required, with the Social Filter.

- The Social Filter operates on a social network of services to compute the strength of the relationships between them. The EventCloud uses a relationship strength threshold or a ranking of the destination services to select the most trustworthy services. In concrete terms, the EventCloud accepts to process (notify) the events that the source service received from target services with which it has high relationship strength whereas it discards the ones received from target services with low relationship strength. The social filter provides an interface to define relationships between services.

- The DiCEPE aims to detect complex events using events from various sources such as federated SOAs, and to share out those new complex events, which have been identified as business events, to enrich the whole event process. Inputs come from the EventCloud that prefilters some irrelevant events with simple operators. Others detections that require correlation between multiple events are let to the DiCEPE. Facts inferred are sent to the SeaCloud that forwards them to the DSB to perform adaptation.

- Monitoring is in charge of collecting monitoring information about services and events. It offers a web-based frontend interface that allows getting information and statistics about running infrastructure in term of services and events. The monitoring component relies on EasierBSM to monitor service providers or event producers. It is part of the process to detect SLO (Service Level Objective)/ELO (Event Level Objective) violation which are prerequisites to ensure QoS.

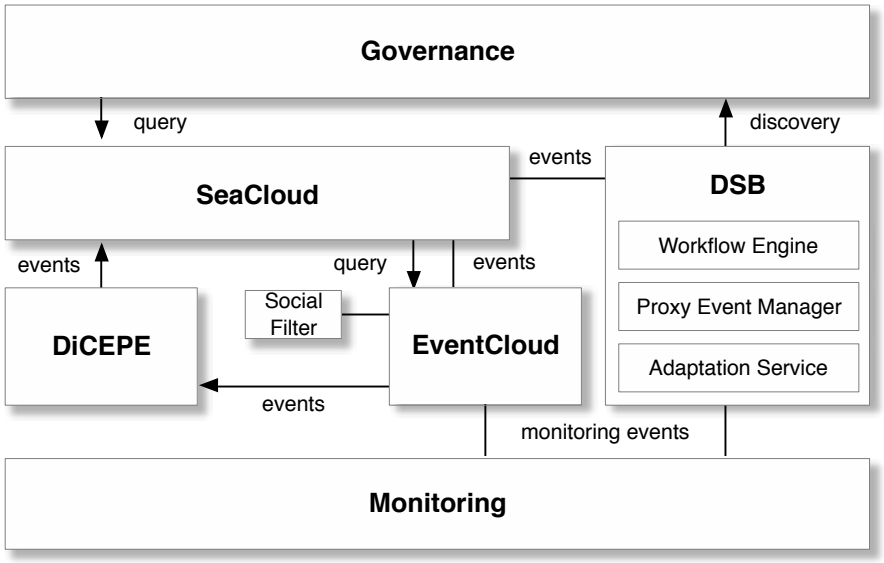


Figure B.1 – Conceptual SocEDA architecture.

Annexe C

Extended Abstract in French

Un intergiciel gérant des événements pour permettre l'émergence d'interactions dynamiques et ubiquitaires dans l'Internet des services.

C.1 Introduction

C.1.1 Motivation

Ces dernières années, le trafic Internet échangé a augmenté de façon vertigineuse. Cette explosion est expliquée par la quantité d'informations générée par les utilisateurs et les nouveaux services qui se développent à un rythme effrainé. Comme l'a déclaré Eric Schmidt en 2010, tous les deux jours maintenant nous créons autant d'informations que ce que nous en avons produit depuis l'aube de la civilisation jusqu'en 2003. A cette date, la quantité d'information représentait déjà quelque chose comme cinq exaoctets de données, a-t-il dit. En outre, avec l'avènement de l'"Internet des objets", concept qui se réfère à des objets identifiables de manière unique communiquant sur Internet, nous n'en sommes probablement qu'au début de cette croissance exponentielle de l'information. Pour exemple, Cisco prévoit que le trafic IP mondial atteindra 1,4 zettaoctets par an en 2017[1].

Cette explosion concernant les informations échangées a donné naissance à un nouveau domaine de la science informatique appelé Data Mining. L'objectif global

[1] `http://goo.gl/dj85Ul`

du processus d'exploration de données est de découvrir des tendances intéressantes dans de grands ensembles de données. Un exemple concret est Google Knowledge Graph [1] qui fournit des informations supplémentaires utiles lorsque vous effectuez une recherche. Un exemple plus récent est le scandale provoqué par le programme de surveillance prisme exploité par l'Agence Nationale de la sécurité Américaine (NSA), dont le but est de collecter et de corréler les méta-données des utilisateurs, à leur insu, pour prévenir les actes terroristes.

Une condition préalable à l'exploration de données est d'agréger des flux d'intérêt et de stocker les données entrantes pour l'analyse. Ces bases de connaissances sont concrétisées par des entrepôts de données qui sont d'énormes dépôts de données créées par l'intégration des données provenant d'une ou plusieurs sources disparates. Lorsque les données source proviennent d'acteurs hétérogènes sur Internet, la construction d'entrepôts de données soulève une question principale : comment filtrer les informations d'intérêt et les corréler avec les autres ? Un élément clé pour répondre à cette question consiste à utiliser des données structurées afin de les rendre analysables et compréhensibles par des machines.

Le mouvement du Web sémantique a suscité un intérêt considérable ces dernières années. Il vise à transformer le Web, à savoir les documents Web, en une base de données gigantesque dans laquelle les ordinateurs peuvent extraire des données d'une manière homogène. Le point intéressant ici est que la communauté du Web sémantique fournit déjà une pile complète de technologies (RDF, SPARQL, RDFS, OWL, etc.) pour répondre à la plupart des problèmes liés à la question précédente, mais principalement dans un environnement synchrone et centralisé. Dans le cadre du projet européen PLAY, l'un des projets dans lequel cette thèse a été développée, nous étudions comment nous pouvons tirer parti du modèle de représentation du Web sémantique et donc de la pile existante pour filtrer, détecter et réagir lorsque des situations intéressantes se produisent. Dans ce contexte, l' objectif du projet PLAY est d'apporter une architecture élastique et fiable pour l'interaction événementiel dynamique et complexe dans les systèmes de services hautement distribués et hétérogènes. Une telle architecture permettra l'échange d'informations omniprésentes entre les services hétérogènes, tout en offrant d'adapter et de personnaliser leur exécution, ce qui mène à la fameuse adaptation de processus dirigée par les situations.

C.1.2 Définition du problème

Dans cette thèse, nous nous concentrons sur deux problématiques clés que l'on peut résumer par les deux questions suivantes : *Comment pouvons-nous stocker efficacement et récupérer des données du Web sémantique dans un environnement distribué ? Comment pouvons-nous faire un filtrage pragmatique et diffuser des événements du Web sémantique pour les utilisateurs ayant des préférences individuelles ?*

La question inhérente au passage à l'échelle qui se pose avec les systèmes distribués a été grandement étudiée ces dernières années et consiste à avoir recours aux réseaux Pair-à-Pair (P2P) qui évitent un point d'accès unique. Toutefois, s'appuyer sur le modèle du Web sémantique, représenté par RDF, soulève plusieurs défis qui ont un impact direct sur la topologie du réseau sous-jacent qui est considéré.

Le premier défi provient du niveau d'expressivité du language SPARQL qui est habituellement utilisé pour récupérer des données RDF. Certains ont peut-être remarqué la similitude lexicographique avec SQL. L'analogie n'est pas une coïncidence. SPARQL est un langage de requête pour les données RDF qui a été développé longtemps après SQL. Bien qu'ils soient tout à fait différents, car ils n'ont pas tout à fait les même objectifs, SPARQL supporte des opérateurs très complexes, ce qui le rend aussi expressif voire plus expressif que SQL. Dans ce manuscrit, nous allons expliquer comment le modèle de données et le langage de requête affectent un grand nombre de choix que nous avons fait tels que la conception de l'architecture P2P, les algorithmes de routage et le stockage des données RDF.

Le deuxième défi est lié au filtrage de données RDF publiées par les éditeurs aux parties intéressées. Comme expliqué dans la partie C1.1**??**, le scénario envisagé dans le projet PLAY est principalement basée sur l'interrogation guidée par les données [2] qui mettent l'accent sur des conditions en quasi-temps réel qui doivent être satisfaites. Une condition préalable est de filtrer l'information d'intérêt mais pas seulement. Les événements doivent être stockés et agir comme un contexte supplémentaire afin de pouvoir réaliser une sorte d'analyse sur les données passées. Les données du Web sémantique ont, encore une fois, un impact significatif sur la façon dont la couche Publier/Souscrire doit être conçue. Cependant, cela génère

une complexité supplémentaire car dans les systèmes à évènements, les entités sont faiblement couplées. Les communications sont effectuées de manière asynchrone, ce qui fournit moins de garanties par rapport au modèle d'échange de messages requête/réponse traditionnel (par exemple de l'absence de garantie de livraison). Dans ce contexte, nous allons voir comment les données RDF peuvent être représentées comme des événements et comment elles diffèrent des traditionnels évènements multi-attributs. En outre, la combinaison entre le filtrage des événements et le stockage soulève plusieurs questions sur l'efficacité et la cohérence du système envisagé : par exemple, comment s'assurer que l'ordre des opérations envoyées depuis un même client est respecté ? à quel type de débit peut-on s'attendre ? comment faire en sorte que les événements soient enregistrés une fois qu'ils ont été notifiés ? ces questions seront examinées et traitées par la suite.

Enfin, un troisième défi à prendre en compte concerne la répartition de charge et la propriété d'élasticité des systèmes distribués modernes dont nous avons l'intention de tirer parti afin d'assurer un certain niveau de performance.

Nous allons voir que le choix que nous avons fait et qui consiste à ne pas utiliser de fonction de hachage afin de supporter des requêtes ou souscriptions plus complexes que la simple correspondance exacte, nous expose à des déséquilibres de charge. Dans la vie courante, la distribution des données est souvent biaisée, mais ici le déséquilibre est accentué par l'une des caractéristiques des données RDF qui implique que certaines valeurs partagent des préfixes communs.

C.1.3 Plan et contribution

La principale contribution de cette thèse est la définition et la mise en œuvre d'un middleware modulaire pour le stockage, la récupération et la diffusion des données RDF et des événements dans des environnements de type cloud. La thèse est structurée autour de trois travaux majeurs organisés en quatres chapitres dédiés dont le contenu est résumé ci-après. Mais avant cela il convient de préciser sommairement le contenu des autres chapitres venant en complément :

- Le **Chapitre 2** donne un aperçu des principaux concepts et technologies auxquels nous nous réferrons tout au long de cette thèse. Tout d'abord, nous introduisons le paradigme Pair-à-Pair. Ensuite, nous attirons l'attention sur

le Web sémantique et nous discutons des principaux avantages d'utiliser la sémantique avant de se concentrer sur le modèle de communication Publier/-Souscrire. Enfin, nous détaillons le middleware ProActive, qui est la principale technologie utilisée pour mettre en œuvre le middleware développé dans le cadre de cette thèse.

- Le **Chapitre 3** présente notre première contribution qui se rapporte à une infrastructure de stockage RDF distribuée, introduite dans [3] et élue meilleur papier de la conférence AP2PS 2011. Une section courte des travaux connexes sur les systèmes décentralisées pour stocker et récupérer des données RDF introduit ce chapitre. Ensuite, nous présentons le populaire protocole P2P CAN qui définie la topologie sous-jacente du réseau P2P sur laquelle nous nous appuyons pour le routage des messages et, indirectement, pour passer à l'échelle. Suivra une discussion sur les choix de conception et les ajustements que nous avons faits en ce qui concerne le protocole CAN avant d'expliquer dans une deuxième partie comment les messages sont acheminés avec nos modifications. Dans une avant-dernière section, nous décrivons en détail comment les données RDF sont indexées dans le réseau P2P et comment les requêtes SPARQL sont exécutées. Enfin, nous fournissons les résultats que nous avons obtenus par l'expérimentation de notre solution sur le banc d'essai Grid'5000.

- Le **Chapitre 4** entre dans les détails de notre deuxième contribution mettant en avant une couche de publier/souscrire pour le stockage et la diffusion des événements RDF. Elle est construite comme une extension sur l'infrastructure mise en place dans le chapitre précédent et s'appuie sur les algorithmes de routage descrits plus tôt. Nous commençons par comparer les solutions existantes et nous expliquons pourquoi les systèmes d'événements basés sur RDF diffèrent des systèmes publier/souscrire traditionnels. Ensuite, nous présentons notre infrastructure publier/souscrire pour événements RDF avec un détail du modèle d'évènements et de souscription, approprié pour les données RDF, que nous proposons. Puis, nous énumérons les différentes propriétés que notre système de publier/souscrire est supposé respecter avant d'entrer dans les détails de deux algorithmes publier/sous-

crire. Leurs caractéristiques et leurs différences sont expliquées, discutées et analysées. Pour conclure, les algorithmes que nous proposons sont évalués dans un environnement distribué avec un maximum de 29 machines. Cette deuxième contribution a été acceptée et présentée durant la conférence Globe 2013 [4].

- Le **Chapitre 5** met en évidence notre troisième contribution qui se réfère à la répartition de charge avec les données RDF. La première section résume comment les solutions de répartition de charge ont évoluées au fil du temps et quelles sont les solutions pour corriger les déséquilibres de charge avec les données RDF. Ensuite, nous avons décrit notre solution en expliquant les différents choix qui sont envisageables et ceux pour lesquels nous avons optés. Notre approche combine des mécanismes standards tels que des protocoles d'échange d'informations par rumeur ou encore l'enregistrement à la volée de statistiques dans le but d'améliorer la distribution des donnés. Dans une dernière section, nous présentons les résultats obtenus pour les évaluations empiriques que nous avons réalisées avec des données réelles.

- Le **Chapitre 6** donne un aperçu de l'intergiciel EventCloud, qui est l'intergiciel développé dans le cadre de cette thèse. Le but de ce chapitre est de donner un aperçu du système du point de vue l'architecture et l'implémentation. Dans un premier temps, nous mettons en évidence les différents composants qui constituent le système. Ensuite, nous résumons ses différentes caractéristiques et nous montrons en quoi l'intergiciel est flexible et modulaire. En particulier, nous voyons comment la modularité joue un rôle important dans l'architecture proposée et quel genre d'avantages cela apporte en ce qui concerne les composants qui forment notre infrastructure. Ensuite, nous nous concentrons sur quelques problèmes d'implémentation auxquels nous avons dû faire face et que nous avons corrigés afin de rendre le système plus efficace et réactif.

- Le **Chapitre 7** conclut la thèse. Il passe en revue les contributions et présente quelques perspectives de recherche et de développement que soulève cette thèse.

Par ailleurs, concernant les contributions qui ont pu être réalisées, nous pouvons noter que l'intergiciel EventCloud a été testé et validé avec les différents scénarios créés dans le cadre du projet PLAY [5, 6, 7, 8, 9, 10, 11]. L'intergiciel EventCloud a également été utilisé et évalué dans d'autres contextes. Par exemple, il offre les briques de base pour distribuer des moteurs CEPs visant à corréler plusieurs événements en temps réel avec d'autres du passé [12]. Une autre application concerne les transferts de données paresseux dans laquelle les événements intègrent des pièces jointes volumineuses qui n'ont pas besoin de transiter par le service d'évènements. Seules les descriptions d'événements sont transmises à l'EventCloud avant d'être diffusées aux parties intéressées. Les attachements joints sont transférés de manière paresseuse et transparente au travers d'un échange direct entre émetteurs d'évènements et souscripteurs [13].

C.2 Résumé développement

Le travail abordé dans ce manuscrit de thèse est développé dans quatre chapitres principaux. Ci-après est précisé un résumé en Français de chacun de ces chapitres.

C.2.1 Stockage RDF distribué

Ce premier chapitre présente une infrastructure de stockage RDF distribuée basée sur réseau P2P structuré. Les uplets RDF sont mappés sur un CAN à quatre dimensions selon la valeur des éléments du uplet considéré. L'espace du réseau P2P est partitionné en zones et chaque pair est responsable d'une zone ainsi que de tous les uplets se trouvant à l'intérieur. Nous n'utilisons pas de fonction de hachage afin de préserver la localité des données. Les requêtes SPARQL sont décomposés en sous-requêtes qui sont exécutées en parallèle. Nous avons validé notre implémentation à l'aide de micro expérimentations. Bien que les opérations de base comme l'ajout de uplets souffrent d'un surcoût, la nature de l'infrastructure distribuée permet des accès concurrent. En substance, nous échangeons des performances pour un meilleur débit.

De toute évidence notre solution présente certains inconvénients. Le premier est que notre approche est sensible à la distribution des données. Comme nous

utilisons l'ordre lexicographique pour indexer les données, lorsque certains uplets RDF partagent le même espace de noms ou préfixes, la probabilité qu'ils se retrouvent sur un même pair est très élevée. Par conséquent, un ou plusieurs pairs peuvent devenir un goulot d'étranglement du système. Pour résoudre ce problème, nous avons présenté des solutions dans le Chapitre 5. Le deuxième inconvénient avec notre solution est lié à l'exécution de requêtes SPARQL. Nous avons décidé de traiter chaque sous-requête indépendamment en parallèle. Toutefois, lorsque les sous-requêtes partagent des variables communes (i.e. exigent une jointure) et retournent des ensembles de uplets avec des tailles qui diffèrent d'un ou plusieurs ordres de grandeur, notre solution nécessite de transporter de pairs en pairs jusqu'à l'initiateur de la requête, plusieurs uplets qui ne seraient pas nécessaires si les sous-requêtes étaient exécutées en série grâce à la construction préalable d'un plan d'exécution séquentiel. Ce problème qui est lié au nombre de résultats intermédiaires à transférer dans le réseau afin de résoudre une requête SPARQL a été mis en évidence de façon empirique avec notre système [110]. Quilitz *et al.* propose dans [111] de construire un plan de requêtes qui exécute des sous-requêtes successivement après avoir été triées dans l'ordre décroissant en fonction du nombre de parties fixes et de leur position (i.e. graphe, sujet, prédicat ou objet d'un motif de quadruplet). En effet, les sous-requêtes avec une multitude de parties fixes sont supposées renvoyer moins de résultats que d'autres qui impliqueraient un plus grand nombre de variables lorsque le jeu de donnée est assez grand. Par conséquent, la sous-requête à exécuter peut exploiter le résultat de la précédente afin de réduire le nombre de résultats intermédiaires. Un pas en avant a été franchi puisque dans [112] les auteurs proposent d'étudier la sélectivité de sous-requêtes (i.e une estimation ou une valeur exacte du nombre de uplets qu'une sous-requête est sensée renvoyer une fois exécutée). De cette façon, un plan de requête optimal peut être défini. Une perspective pourrait être de combiner notre solution avec celle précédemment décrite pour réaliser un plan de requête optimal. L'idée est de toujours exécuter en parallèle les sous-requêtes avec une sélectivité faible ou bien d'exécuter celles qui, avec un arrangement intelligent, ne peuvent pas réduire la consommation de bande passante induite par le transfert abondant de résultats intermédiaires. Aussi, un autre point qui affecte les performances systèmes concerne la latence de routage des messages. Pour améliorer le temps d'exécution,

la structure logique du réseau P2P CAN pourrait être revue afin de prendre en compte la localité physique des pairs qui le compose, ceci afin de rapprocher les nœuds proches physiquement de façon à réduire la latence. Ali *et al.* ont montrés dans [113] qu'un système P2P structuré qui prend en compte la localité physique des machines, et qui ajoute quelques raccourcis supplémentaires, peut améliorer les performances d'un facteur de deux dans un contexte RDF.

C.2.2 Publier/Souscrire RDF distribué

Ce second chapitre introduit une infrastructure publier/souscrire basée sur le modèle de données RDF et le modèle de filtre SPARQL. Les souscripteurs peuvent exprimer leur intérêts en utilisant un sous-ensemble du langage SPARQL et les évènements sont publiés comme données RDF. Nous nous appuyons sur une indexation multi-dimensionnelle et l'ordre lexicographique pour distribuer les publications et les souscriptions sur la structure logique associée au réseau P2P.

	Élément routé	**Matching**	**Doublons**	**Happen-Before**
CSMA	Quadruplets individuels	Multiple, Séquentiel et Reconstruction	Oui, filtrage requis	Imposé
OSMA	*Compound Event* complet	Unique étape	Non	Exige CSMA

TABLE C.1 – Comparaison des algorithmes publier/souscrire proposés.

Contrairement à la grande majorité des systèmes existants, notre solution ne nécessite pas d'indexer de multiple fois la même publication, limitant ainsi l'espace de stockage nécessaire. Nous avons proposé deux algorithmes pour tester la correspondance entre les souscriptions et les événements publiés. Le premier, CSMA, est basé sur une approche séquentielle. Cela limite la bande passante utilisée lors

des publications en contre partie d'un temps plus long pour détecter les souscriptions qui sont vérifiées. Il peut également gérer les problèmes d'ordonnancement des requêtes entre des publications et des souscriptions qui proviennent d'un même hôte. Le second, OSMA, utilise une approche totalement distribuée ce qui conduit à de bonnes performances mais un temps requis légèrement plus important dans le processus de publication. Pour résumer, les différentes propriétés de ces deux algorithmes sont présentées dans le tableau C.1. Ils ont été testés expérimentalement en terme de débit par seconde et de passage à l'échelle.

C.2.3 Répartition de charge RDF distribuée

Ce troisième chapitre présente et analyse brièvement deux stratégies pour répartir équitablement les données RDF sur notre version révisée du réseau CAN. L'idée centrale est de partager les surcharges entre pairs en divisant les zones des pairs non pas en leur milieu comme suggéré par le protocole CAN par défaut mais au point qui équilibre la distribution des données RDF. Cela est rendu possible par l'enregistrement des valeurs de barycentre de termes RDF par dimension. Ensuite, les différentes stratégies qui sont proposées diffèrent principalement par la façon dont les déséquilibres sont détectés. La première suppose une connaissance globale tandis que la seconde s'appuie sur des informations échangées entre pairs périodiquement. Nos expériences ont montré que la seconde stratégie donne de meilleurs résultats que la première. Bien que la solution que nous proposons soit loin d'être idéale dans le sens ou les données RDF ne sont pas aussi bien distribuées que ce qu'elles pourraient l'être, les stratégies proposées améliore la distribution, l'impliquation des pairs et donc le débit de sortie quand la couche publier/souscrire est utilisée.

Il convient de mentionner que la solution présentée est un travail inachevé. Plusieurs points nécessiteraient d'être examinés plus en profondeur et des expérimentations plus intensives devraient menées. De plus, de nombreux aspects de la solution présentée pourraient être améliorés. Par exemple, le protocole de diffusion de charge des pairs pourrait être optimisé en mettant en œuvre ce qui est proposé dans la Section 5.2. En outre, avant d'allouer de nouveaux pairs, la délocalisation sur des pairs existants devrait être envisagée. Une orientation supplémentaire

pourrait être de considérer plusieurs critères tels que la charge d'exécution des requêtes synchrones, des souscriptions, la consommation CPU voire la bande passante utilisée. Puisque notre modèle de répartition de charge a été conçu avec l'idée de prendre en charge plusieurs critères indépendants, en ajouter de nouveaux ne devrait pas être difficile.

C.2.4 Implémentation

Finalement, le quatrième chapitre détaille le système mis en œuvre. Pour valider et évaluer notre solution avec les algorithmes proposés, nous avons conçu et implémenté l'intergiciel EventCloud qui est une application fournissant un service distribué de stockage d'informations par dessus un réseau CAN à 4-dimensions pour stocker et récupérer des quadruplets à l'aide de SPARQL mais également pour gérer des événements représentés en RDF. L'intergiciel EventCloud a été implémenté en Java. Il s'appuie sur la bibliothèque de programmation ProActive et son extension des objets multi-actifs afin de respectivement distribuer les composants et tirer parti des processeurs multi-core en exécutant, lorsque cela est possible, des requêtes en parallèle à d'autres. Une attention particulière a été portée sur la modularité de l'architecture, améliorant ainsi la réutilisation et la flexibilité. En outre, les goulots d'étranglement de performance ont été étudiés et des solutions proposées.

Un autre apport de notre solution est son intégration dans les plates-formes développées dans le cadre des projets PLAY et SocEDA dont les objectifs et architecture respectifs sont présentés dans l'annexe A and B. Les critères évalués ont été la performance, la stabilité et la facilité d'utilisation [158, 159]. Il est à noter que certaines fonctionnalités ont été encouragées par les projets dans lesquels nous avons été impliqués. Toutefois, leur utilité n'est pas uniquement limité aux projets. Cela comprend par exemple nos efforts pour fournir un traducteur XML/RDF ou bien encore le temps passé à rendre le déploiement du réseau P2P EventCloud plateforme agnostique en réutilisant et en étendant les abstractions fournies par le déploiement GCM et ProActive. Un autre exemple est le service Web EventClouds, prototypé avec Flask[2], que nous avons introduit afin de pou-

[2] `http://flask.pocoo.org`

voir gérer plusieurs instances EventCloud, ce qui vise à rendre la gestion plus pratique pour les administrateurs tout en augmentant l'interopérabilité. Cette application Web permet de créer, supprimer, visualiser la liste des EventClouds et des proxies disponibles par l'intermédiaire de messages SOAP envoyés au service Web. Pour atteindre son but, ce dernier s'appuie sur plusieurs abstractions telles qu'un registre EventClouds afin de maintenir une référence sur les instances disponibles et leur configuration de déploiement associée, mais aussi un fournisseur de nœuds pour abstraire la manière et l'infrastructure à partir desquelles les nœuds sont récupérés. Évidemment ces outils de déploiement et de gestion peuvent être réutilisés pour gérer les instances EventCloud à l'intérieur d'une organisation ou entre plusieurs organisations. Plus de détails sur les fonctionnalités supplémentaires construites autour du noyau de l'intergiciel EventCloud sont donnés dans les livrables de projets [9, 160, 161, 162, 163].

La base du code qui constitue le noyau de l'EventCloud contient approximativement quarante cinq mille lignes de code ainsi que trente mille lignes de commentaires et plus de 260 tests unitaires. L'ensemble est réparti entre six cent cinquante fichiers accessibles publiquement à l'adresse `http://eventcloud.inria.fr`.

C.3 Conclusion

C.3.1 Résumé

RDF est devenu un modèle de données pertinent pour la description et la modélisation de l'information sur le Web tout en restant relativement simple et intuitif. Cependant, gérer et de stocker des informations RDF de manière synchrone ou asynchrone a soulevé de nombreuses questions qu'en au problème de passage à l'échelle dans un contexte distribué. Le résultat principal de cette thèse est un intergiciel dédié au stockage, à la récupération synchrone mais aussi à la diffusion sélective et asynchrone des données RDF.

Notre première contribution concerne la conception d'une infrastructure distribuée pour le stockage et le traitement de données RDF et de requêtes SPARQL dans un contexte synchrone en utilisant le modèle requête/réponse traditionnel. L'architecture est basée sur une topologie logique CAN à quatre dimensions où

les uplets RDF sont indexés selon l'ordre lexicographique de leurs éléments. Le système n'utilise pas de fonctions de hachage et évite ainsi de stocker les mêmes informations plusieurs fois. En outre, l'indexation de données selon l'ordre lexicographique permet de traiter de manière efficace les requêtes à intervalles. Bien que les opérations de base comme l'ajout de données RDF souffrent d'une surcharge par rapport à une solution centralisée, la nature distribuée de l'infrastructure permet des accès concurrents. Les requêtes SPARQL sont ainsi évaluées en les décomposant en sous-requêtes qui sont exécutées en parallèle.

La deuxième contribution de cette thèse est une couche de publier/souscrire pour le stockage et la diffusion sélective des événements RDF. Nous avons proposé un modèle de données et de souscription respectivement basé sur une extension de RDF et un sous-ensemble de SPARQL. En outre, nous avons conçu deux algorithmes publier/souscrire, à savoir CSMA et OSMA, chacun visant des besoins différents. Le premier, CSMA, inspiré par Liarou *et al.*, effectue la concordance entre les évènements et les souscriptions en séquentiel mais est en mesure de fixer les problèmes d'ordonnancement des requêtes entre des publications et des souscriptions qui proviennent d'un même hôte. Au contraire, le second, OSMA, utilise une approche totalement distribuée permettant de faire correspondre les publications et les abonnements directement en une seule étape, ce qui conduit à de meilleures performances mais dans ce cas la consommation de bande passante est légèrement plus importante. Les expériences menées ont montré que les deux algorithmes sont complémentaires en fonction du scénario considéré.

Notre troisième contribution concerne la répartition de charge. Les systèmes distribués RDF souffrent d'une répartition inégale des termes RDF. Les uplets avec des termes RDF qui apparaissent plus fréquemment que d'autres sont indexés sur plusieurs nœuds ce qui engendre des surcharges. Dans ce contexte, nous avons présenté et évalué des stratégies pour améliorer la répartition des données RDF entre les différents pairs de la version révisée du réseau P2P CAN. La solution que nous proposons combine des techniques existantes basées sur des protocoles de diffusion d'information par rumeur et l'enregistrement de statistiques associées aux données. La première technique nous permet de diffuser la charge mesurée sur chaque pair afin que la décision de répartition de charge soit prise en comparant la charge de pairs avec une charge moyenne du système calculé en fonction des

mesures échangées. Une fois qu'un déséquilibre est détecté, les statistiques enregistrées sur les données RDF par dimension nous permettent de décider comment la zone de responsabilité de pairs peut être fractionnée pour répartir équitablement le déséquilibre. Des expériences ont montré que la surcharge induite par l'enregistrement de données statistiques à la volée reste acceptable et le fait de forcer de nouveaux pairs à joindre les pairs surchargés en divisant leur zone sur la base des informations enregistrées peut grandement diminuer les déséquilibres de charge.

Enfin, nous avons consacré des efforts à fournir au niveau de l'implémentation un intergiciel flexible et modulaire avec des séparations claires entre les différents éléments logiciels qui le compose.

C.3.2 Perspectives

Dans ce qui suit nous proposons quelques perspectives qui pourraient être explorées, en particulier quelques pistes pour améliorer l'efficacité et la valeur ajoutée de l'intergiciel proposé.

Optimiser l'évaluation des requêtes et des souscriptions

Comme nous avons commencé à le mettre en évidence dans leurs chapitres respectifs, l'évaluation des requêtes SPARQL synchrones mais aussi les algorithmes de correspondance de la couche publier/souscrire pourraient profiter des domaines de recherche suivants.

Améliorer l'exécution du plan de requête Le chapitre 3 a montré que l'exécution des requêtes SPARQL distribué dépend de la complexité des requêtes et de l'espace de recherche. Notre approche consiste à décomposer une requête en sous-requêtes et d'exécuter les sous-requêtes de façon indépendante et en parallèle. Toutefois, lorsque les sous-requêtes partagent des variables communes (i.e. nécessitent une jointure) et retournent des ensembles de uplets avec des tailles qui diffèrent d'un ou plusieurs ordres de grandeur, notre solution nécessite de transporter de pairs en pairs jusqu'à l'initiateur de la requête. Plusieurs uplets ne seraient alors pas nécessaires si les sous-requêtes étaient exécutées en série en construisant auparavant un plan d'exécution séquentiel. Par conséquent, une piste serait

de construire des plans d'exécution de requêtes qui soient appropriés pour les requêtes SPARQL impliquant des sous-requêtes partageant des variables communes. Néanmoins, notre approche parallèle reste avantageuse pour les sous-requêtes indépendantes. Ainsi, une avancée consisterait à générer des plans de requêtes en combinant l'exécution en parallèle de sous-requêtes indépendantes avec l'exécution en série de celles qui nécessitent des jointures. Les travaux effectués dans [111] et [112] pour améliorer respectivement l'exécution des plans d'exécution des requêtes en réorganisant les triple patterns, mais aussi estimer leur sélectivité, sont probablement des ressources intéressantes à consulter. Pour améliorer davantage les transferts de données, des techniques de compression pourraient être appliquées [164]. Par exemple, afin d'améliorer le temps d'exécution, il pourrait être envisagé de remodeler la structure logique du réseau P2P CAN en prenant en compte la localité physique des pairs qui le compose, ceci afin de rapprocher les nœuds proches physiquement pour réduire la latence. Ali *et al.* ont montré dans [113] qu'un système P2P structuré qui prend en compte la localité physique des machines, et qui ajoute quelques raccourcis supplémentaires, peut améliorer les performances d'un facteur de deux dans un contexte RDF.

Synthétiser les souscriptions Concevoir un système publier/souscrire soulève de nombreuses questions. Selon le contexte dans lequel il est utilisé, un défi majeur est de gérer un nombre important de souscriptions. Étant donné que les souscriptions sont enregistrées sur les pairs, lorsque chaque souscription est gérée indépendamment, plus le nombre est important, plus le temps de traitement augmente. Pour remédier à ce problème une solution est de synthétiser un ensemble de souscriptions en une seule [165]. Le but est de trouver des relations de subsomption entre les intérêts de plusieurs souscriptions afin de les combiner en une nouvelle résumée réduisant ainsi la consommation de bande passante, le surplus de stockage et le nombre de messages échangés entre pairs pour effectuer la correspondance contre les événements. Plusieurs approches ont été proposées ces dernières années [166, 167, 168]. L'application d'une telle technique à notre solution permettrait d'améliorer encore plus ses performances.

Améliorer la fiabilité et la disponibilité

Même si les algorithmes de routage de l'intergiciel proposé ont été conçus en ayant à l'esprit la gestion de la tolérance aux pannes pour éviter une refonte complète à l'avenir, la gestion de la fiabilité et de la disponibilité n'a pas été implémentée. Notre intergiciel tirerait un avantage intéressant d'une solution permettant un redémarrage en toute sécurité en cas de défaillance des pairs, cela impliquant entre autre de supporter la tolérance aux pannes au niveau des objets multi-actifs. En ce qui concerne ce dernier point, plus de détails seront fournis dans la thèse qu'est en train de réaliser Justine Rochas. Un bon point de départ pour améliorer la disponibilité, la fiabilité, mais aussi la performance des opérations de lecture et donc l'évaluation des requêtes SPARQL synchrones pourrait être d'étudier la réplication. Cela pourrait se faire par l'étude de solutions de réplication existantes telles que celles proposées dans le document CAN original [107], les auteurs de Meghdoot [122] ou encore par la recherche de nouvelles moins sensibles à l'arrivée et aux départs fréquents de pairs, comme proposé dans [169].

Raisonner sur les données RDF

RDF est un modèle de données idéal pour l'écriture d'informations destinées à être échangées à l'échelle mondiale. Toutefois, capturer la sémantique ou la signification des informations nécessite d'autres normes de la pile du Web sémantique comme RDFS et/ou OWL (cf. Figure 2.3) . Ces derniers permettent de définir des vocabulaires en définissant les éléments utilisés dans une application, leur domaine, leur type, leurs relations, mais aussi les contraintes possibles concernant leur utilisation. En d'autres termes, les technologies RDFS et OWL fournissent une base solide pour comprendre et déduire de nouvelles informations pertinentes. Cet objectif est généralement matérialisé en calculant la fermeture transitive d'un graphe RDF. Cela consiste à rendre toute information implicite, explicite en appliquant les règles RDFS/OWL sur les données RDF jusqu'à ce qu'aucune nouvelle donnée ne soit dérivée. Interpréter les vocabulaires RDFS et OWL avec les règles d'inférence qui leur sont associées de manière distribuée, tout en passant à l'échelle, reste un défi et serait une perspective intéressante à creuser afin de pouvoir exploiter le potentiel réel de RDF [170, 171, 172].

Pour conclure, un autre point de vue pourrait être d'explorer une solution basée sur une version légèrement adaptée du modèle MapReduce afin de charger des événements RDF et traiter des requêtes continues en utilisant SPARQL [173], dans le but de comparer les performances avec notre solution. Plus de détails seront fournis dans la thèse de Sophie Ge Song. Dans une toute autre direction, il pourrait être envisagé d'étudier l'applicabilité des bases de données de type graphe (par exemple Trinity [174] ou Stardog[3]) pour le stockage, la récupération et la diffusion sélective de données RDF à très grande échelle.

[3] `http://stardog.com`

Bibliography

[1] Amit Singhal. Introducing the knowledge graph: things, not strings. *Official google blog*, 2012. URL: `http://googleblog.blogspot.fr/2012/05/introducing-knowledge-graph-things-not.html` (cited on pp. 2, 206).

[2] Darko Anicic, Paul Fodor, Roland Stuhmer, and Nenad Stojanovic. Event-driven approach for logic-based complex event processing. In *Computational Science and Engineering*. Volume 1. IEEE, 2009, pages 56–63 (cited on pp. 3, 207).

[3] Imen Filali, Laurent Pellegrino, Francesco Bongiovanni, Fabrice Huet, Françoise Baude, et al. Modular P2P-based approach for RDF data storage and retrieval. In *Proceedings of the international conference on Advances in P2P Systems*, 2011 (cited on pp. 5, 209).

[4] Laurent Pellegrino, Fabrice Huet, Françoise Baude, and Amjad Alshabani. A distributed publish/subscribe system for RDF data. In, *Proceedings of the international conference on Data Management in Cloud, Grid and P2P Systems (Globe)*. Springer, 2013 (cited on pp. 5, 210).

[5] G. Mentzas, D. Apostolou, Yannis Verginadis, V. Tsalikis, Nenad Stojanovic, Roland Stuehmer, Jean-Pierre Lorré, Christophe Hamerling, Françoise Baude, and Francesco Bongiovanni. D1.1 State of the art. Project Deliverable PLAY. 2011. URL: `http://play-project.eu/documents/viewdownload/3/13` (visited on 08/18/2013) (cited on pp. 6, 211).

[6] Yiannis Verginadis, Nikos Papageorgiou, Yannis Patiniotakis, Aurélie Charles, Matthieu Lauras, Frédérick Benaben, Anne-Marie Barthe, Sébastien Truptil, Roland Stuehmer, Nenad Stojanovic, Françoise Baude, Fabrice Huet, Francesco Bongiovanni, Laurent Pellegrino, Christophe Hamerling, Philippe

Gibert, Osvaldo Cocucci, Bratislav Stoiljkovic, and Zivota Jankovic. D1.3 Requirements analysis. Project Deliverable PLAY. 2011. URL: http://play-project.eu/documents/viewdownload/3/15 (visited on 08/18/2013) (cited on pp. 6, 211).

[7] Roland Stuehmer, Ljiljana Stojanovic, Nenad Stojanovic, Yiannis Verginadis, Laurent Pellegrino, and Christophe Hamerling. D1.4 PLAY conceptual architecture. Project Deliverable PLAY. 2011. URL: http://play-project.eu/documents/viewdownload/3/19 (visited on 08/18/2013) (cited on pp. 6, 211).

[8] Françoise Baude, Francesco Bongiovanni, Laurent Pellegrino, and Vivien Quema. D2.1 Requirements EventCloud. Project Deliverable PLAY. 2011. URL: http://play-project.eu/documents/summary/3/20 (visited on 08/18/2013) (cited on pp. 6, 211).

[9] Iyad Alshabani, Françoise Baude, Laurent Pellegrino, Bastien Sauvan, Philippe Gibert, Christophe Hamerling, Yiannis Verginadis, Sébastien Truptil, and Roland Stuehmer. D2.5.1 PLAY federated middleware specification and implementation v1. Project Deliverable PLAY. 2012. URL: http://play-project.eu/documents/summary/3/131 (visited on 08/18/2013) (cited on pp. 6, 189, 211, 216).

[10] Laurent Pellegrino, Françoise Baude, Iyad Alshabani, and Roland Stuehmer. D3.3 PLAY platform quality of service. Project Deliverable PLAY. 2013. URL: http://play-project.eu/documents/summary/3/238 (visited on 12/10/2013) (cited on pp. 6, 211).

[11] Christophe Hamerling, Laurent Pellegrino, Roland Stuehmer, Philippe Gibert, and Yiannis Verginadis. D5.1.2 Integrated PLAY platform & platform manual v1. Project Deliverable PLAY. 2012. URL: http://play-project.eu/documents/summary/3/201 (visited on 08/18/2013) (cited on pp. 6, 211).

[12] Laurent Pellegrino, Iyad Alshabani, Françoise Baude, Roland Stuehmer, and Nenad Stojanovic. An approach for efficiently combining real-time and past events for ubiquitous business processing. In *International workshop*

on Semantic Business Process Management (SBPM), 2012 (cited on pp. 6, 89, 211).

[13] Quirino Zagarese, Gerardo Canfora, Eugenio Zimeo, Iyad Alshabani, Laurent Pellegrino, and Françoise Baude. Efficient data-intensive event-driven interaction in SOA. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. ACM, 2013, pages 1907–1912 (cited on pp. 6, 211).

[14] Mark Jelasity and A-M Kermarrec. Ordered slicing of very large-scale overlay networks. In *Peer-to-Peer Computing*. IEEE, 2006, pages 117–124 (cited on p. 8).

[15] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing*. IEEE, 2001, pages 99–100 (cited on p. 9).

[16] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: a distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*. Springer, 2001, pages 46–66 (cited on p. 9).

[17] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005 (cited on p. 9).

[18] Bram Cohen. The BitTorrent protocol specification. 2008 (cited on pp. 9, 11).

[19] Sebastian Ertel. Unstructured P2P networks by example: Gnutella 0.4, Gnutella 0.6. URL: `http://ra.crema.unimi.it/turing/materiale/admin/corsi/sistemi/lezioni/m3/m3_u2_def/ceravolo_file2.pdf` (visited on 07/30/2013) (cited on p. 9).

[20] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*. Volume 31(4). ACM, 2001 (cited on pp. 9, 44, 54, 80).

[21] 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. 9, 41, 55, 66, 81).

[22] 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. 9, 55, 66, 79, 127).

[23] 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. 10).

[24] 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. 10, 130, 139).

[25] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: support of range query and cover query over DHT. In *International workshop on Peer-to-Peer Systems (IPTPS)*, 2006 (cited on p. 10).

[26] Luis Garces-Erice, Ernst W Biersack, Keith W Ross, Pascal A Felber, and Guillaume Urvoy-Keller. Hierarchical peer-to-peer systems. *Parallel Processing Letters*, 13(04):643–657, 2003 (cited on p. 11).

[27] 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. 11).

[28] Ming Xu, Shuigeng Zhou, and Jihong Guan. A new and effective hierarchical overlay structure for peer-to-peer networks. *Computer Communications*, 34(7):862–874, 2011 (cited on p. 11).

[29] Mourad Amad, Ahmed Meddahi, Djamil Aissani, and Zonghua Zhang. HPM: a novel hierarchical peer-to-peer model for lookup acceleration with provision of physical proximity. *Journal of Network and Computer Applications*, 2012 (cited on p. 11).

[30] LLC Napster. Napster, 2001. URL: `http://www.napster.com` (visited on 07/30/2013) (cited on p. 11).

[31] Jaan Tallinn Ahti Heinla Priit Kasesalu. KaZaA. 2011. URL: `http://www.kazaa.com` (visited on 07/30/2013) (cited on p. 11).

[32] 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. 12).

[33] BP Abbott, R Abbott, R Adhikari, P Ajith, B Allen, G Allen, RS Amin, SB Anderson, WG Anderson, MA Arain, et al. Einstein@home search for periodic gravitational waves in early s5 ligo data. *Physical review d*, 80(4):042003, 2009 (cited on p. 12).

[34] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf` (visited on 07/30/2013) (cited on p. 12).

[35] 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. 13, 49, 52, 79).

[36] 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 (SOSP)*. Volume 7, 2007, pages 205–220 (cited on p. 13).

[37] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011 (cited on p. 13).

[38] Cosmin Arad, Tallat Mahmood Shafaat, and Seif Haridi. CATS: linearizability and partition tolerance in scalable and self-organizing key-value stores. Technical report. Swedish Institute of Computer Science, 2012. URL: `http://soda.swedish-ict.se/5260/1/cats-sics-tr-2012-04.pdf` (visited on 08/19/2013) (cited on p. 13).

[39] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific American*, 284(5):28–37, 2001 (cited on p. 13).

[40] 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 08/07/2013) (cited on p. 14).

[41] Ora Lassila and Ralph R Swick. Resource description framework (RDF) model and syntax specification, 1999 (cited on p. 15).

[42] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The semantic web revisited. *Intelligent Systems*, 21(3):96–101, 2006 (cited on p. 16).

[43] Patrick Hayes and Brian McBride. RDF semantics. 2004. URL: http://www.w3.org/TR/rdf-mt/ (visited on 12/12/2013) (cited on p. 16).

[44] 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. 18).

[45] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *Proceedings of the international conference on World Wide Web*. ACM, 2002, pages 592–603 (cited on p. 18).

[46] Jeen Broekstra and Arjohn Kampman. SeRQL: a second generation RDF query language. In *Proceedings of SWAD-Europe workshop on Semantic Web Storage and Retrieval*, 2003, pages 13–14 (cited on p. 18).

[47] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of SquishQL, a simple RDF query language. In, *The Semantic Web - ISWC*, pages 423–435. Springer, 2002 (cited on p. 18).

[48] Eric Prud'Hommeaux, Andy Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008. URL: http://www.w3.org/TR/rdf-sparql-query (visited on 08/08/2013) (cited on p. 20).

[49] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: an XML query language. *W3C working draft*, 12, 2003 (cited on p. 21).

[50] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. *Continuous queries over append-only databases*. Volume 21(2). ACM, 1992 (cited on p. 21).

[51] Timothy H Harrison, David L Levine, and Douglas C Schmidt. The design and performance of a real-time CORBA event service. *ACM SIGPLAN Notices*, 32(10):184–200, 1997 (cited on p. 24).

[52] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *ACM SIGOPS Operating Systems Review*. Volume 27. (5). ACM, 1994, pages 58–68 (cited on p. 24).

[53] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. SCRIBE: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications*, 20(8):1489–1499, 2002 (cited on pp. 25, 82).

[54] Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H Payberah, and Seif Haridi. Vitis: a gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks. In *International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pages 746–757 (cited on p. 25).

[55] Peter R Pietzuch and Jean M Bacon. Hermes: a distributed event-based middleware architecture. In *International conference on Distributed Computing Systems Workshops*. IEEE, 2002, pages 611–618 (cited on pp. 25, 79).

[56] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *Software Engineering*, 27(9):827–850, 2001 (cited on pp. 25, 81).

[57] Patrick Thomas Eugster. Type-based publish/subscribe. PhD thesis. Swiss Federal Institute of Technology in Lausanne, 2001 (cited on p. 26).

[58] Denis Caromel, Wilfried Klauser, and Julien Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, 1998 (cited on p. 26).

[59] Francoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In, *Euro-Par Parallel Processing*, pages 644–653. Springer, 2005 (cited on p. 26).

[60] Françoise Baude, Denis Caromel, Nicolas Maillard, and Elton Mathias. Hierarchical mpi-like programming using GCM components as implementation support. In *CoreGRID workshop: Grid Systems, Tools and Environments*, 2006 (cited on p. 26).

[61] Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssiere. Communicating mobile active objects in java. In *High Performance Computing and Networking*. Springer, 2000, pages 633–643 (cited on p. 26).

[62] Isabelle Attali, Denis Caromel, and Arnaud Contes. Deployment-based security for grid applications. In, *International Conference on Computational Science*, pages 526–533. Springer, 2005 (cited on p. 26).

[63] Françoise Baude, Denis Caromel, Fabrice Huet, Lionel Mestre, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the international symposium on High Performance Distributed Computing (HPDC)*. IEEE, 2002, pages 93–102 (cited on p. 26).

[64] Denis Caromel, Cédric Dalmasso, Christian Delbé, Fabrice Fontenoy, and Oleg Smirnov. OW2 proactive parallel suite: building flexible enterprise clouds. *ERCIM News*, 2010(83):38–39, 2010 (cited on p. 27).

[65] Françoise Baude, Alexandre Bergel, Denis Caromel, Fabrice Huet, Olivier Nano, et al. IC2D: interactive control and debugging of distribution. In, *Large-Scale Scientific Computing*, pages 193–200. Springer, 2001 (cited on p. 27).

[66] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993 (cited on p. 30).

[67] Denis Caromel and Ludovic Henrio. *A theory of distributed objects: asynchrony, mobility, groups, components*. Springer, 2005 (cited on p. 30).

[68] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. In *Coordination Models and Languages*. Springer, 2013, pages 90–104 (cited on p. 31).

[69] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: a grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1-2):5–24, 2009 (cited on p. 33).

[70] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006 (cited on p. 33).

[71] Françoise Baude, Ludovic Henrio, and Paul Naoumenko. A component platform for experimenting with autonomic composition. In *Proceedings of the international conference on Autonomic Computing and Communication Systems*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, page 8 (cited on p. 34).

[72] Francoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In *Cluster Computing and the Grid (CCGRID)*. IEEE, 2007, pages 599–610 (cited on p. 34).

[73] Imen Filali. Improving resource discovery in P2P systems. PhD thesis. University of Nice-Sophia Antipolis, 2011. URL: `http://www.theses.fr/2011NICE4012` (cited on pp. 37, 54).

[74] Stephen Harris and Dr Nicholas Gibbins. 3store: efficient bulk RDF storage, 2003. URL: `http://www.aktors.org/technologies/3store` (visited on 08/23/2013) (cited on p. 39).

[75] Kevin Wilkinson, Craig Sayers, Harumi A Kuno, Dave Reynolds, et al. Efficient RDF storage and retrieval in Jena2. In *International workshop on Semantic Web and Databases (SWDB)*. Volume 3, 2003, pages 131–150 (cited on p. 39).

[76] 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. 39).

[77] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: managing voluminous RDF description bases. In *SemWeb*, 2001 (cited on p. 39).

[78] 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 pp. 39, 165).

[79] 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 pp. 39, 40).

[80] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the international conference on Very large data bases (VLDB)*. VLDB Endowment, 2007, pages 1150–1160 (cited on p. 40).

[81] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008 (cited on p. 40).

[82] Andy Seaborne. Jena TDB. 2009. URL: `http://jena.apache.org/documentation/tdb` (visited on 08/22/2013) (cited on pp. 40, 170).

[83] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010 (cited on p. 40).

[84] Pingpeng Yuan, Pu Liu, H Jin, W Zhang, and L Liu. TripleBit: a fast and compact system for large scale RDF data. *Very Large Data Base (VLDB)*, 6(7):517–528, 2013 (cited on p. 41).

[85] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *Proceedings of the international conference on World Wide Web*. ACM, 2002, pages 604–615 (cited on p. 41).

[86] Peter Haase, Jeen Broekstra, Marc Ehrig, Maarten Menken, Peter Mika, Mariusz Olko, Michal Plechawski, Pawel Pyszlak, Björn Schnizler, Ronny Siebes, et al. Bibster: a semantics-based bibliographic peer-to-peer system. In, *The Semantic Web - ISWC*, pages 122–136. Springer, 2004 (cited on p. 41).

[87] 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. 41).

[88] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: a family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011 (cited on p. 41).

[89] 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. 41, 42, 48, 66).

[90] 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. 41, 81, 128).

[91] 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. 44).

[92] 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. 45, 127).

[93] Felix Heine. Scalable P2P based RDF querying. In *Proceedings of the international conference on Scalable information systems*. ACM, 2006, page 17 (cited on p. 46).

[94] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970 (cited on p. 46).

[95] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating conjunctive triple pattern queries over large structured overlay networks. In, *The Semantic Web - ISWC*, pages 399–413. Springer, 2006 (cited on p. 47).

[96] 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. 49).

[97] Andreas Harth and Stefan Decker. Optimized index structures for querying RDF from the web. In *Third Latin American Web Congress*. IEEE, 2005, 10–pp (cited on p. 50).

[98] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011 (cited on p. 52).

[99] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly, 2013 (cited on pp. 52, 53).

[100] Jans Aasman. Allegro graph: RDF triple database. Technical report. Franz Incorporated, 2006. URL: `http://www.franz.com/agraph/allegrograph/` (visited on 10/14/2013) (cited on pp. 52, 54).

[101] Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project*, 2008. URL: `https://hadoop.apache.org/docs/r0.19.0/hdfs_design.pdf` (visited on 11/29/2013) (cited on p. 53).

[102] Raghava Mutharaju, Sherif Sakr, Alessandra Sala, and Pascal Hitzler. D-SPARQ: distributed, scalable and efficient RDF query engine. In *International Semantic Web Conference (Posters & Demos)*, 2013, pages 261–264 (cited on p. 53).

[103] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *Proceedings of the international conference on World Wide Web*. ACM, 2012, pages 397–400 (cited on p. 53).

[104] Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan Sequeda, and Marcin Wylot. NoSQL databases for RDF: an empirical evaluation. In *The Semantic Web - ISWC*. Springer, 2013 (cited on p. 53).

[105] Bryan Thompson. Bigdata. 2004. URL: `http://www.bigdata.com` (visited on 10/15/2013) (cited on p. 54).

[106] Francesco Bongiovanni and Ludovic Henrio. A mechanized model for CAN protocols. In *Fundamental Approaches to Software Engineering (FASE)*. Vittorio Cortellessa and Dániel Varró, editors. Volume 7793. In Lecture Notes in Computer Science. Springer, 2013, pages 266–281 (cited on p. 63).

[107] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application level multicast using content-addressable networks. In, *Networked Group Communication*, pages 14–29. Springer, 2001 (cited on pp. 63, 66, 194, 220).

[108] 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 p. 64).

[109] 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. 70).

[110] Maeva Antoine, Françoise Baude, and Fabrice Huet. Évaluation d'une architecture de stockage RDF distribuée. In *23èmes journées francophones d'Ingénierie des Connaissances*, 2012. URL: `http://hal.inria.fr/hal-00908461` (visited on 09/09/2013) (cited on pp. 75, 212).

[111] Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In, *The Semantic Web: Research and Applications*, pages 524–538. Springer, 2008 (cited on pp. 75, 193, 212, 219).

[112] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the international conference on World Wide Web*. ACM, 2008, pages 595–604 (cited on pp. 75, 193, 212, 219).

[113] Liaquat Ali, Thomas Janson, Georg Lausen, and Christian Schindelhauer. Effects of network structure improvement on distributed RDF querying. In, *Data Management in Cloud, Grid and P2P Systems*, pages 63–74. Springer, 2013 (cited on pp. 75, 194, 213, 219).

[114] Umeshwar Dayal, Barbara Blaustein, Alex Buchmann, Upen Chakravarthy, Meichun Hsu, R Ledin, Dennis McCarthy, Arnon Rosenthal, Sunil Sarin, Michael J. Carey, et al. The Hipac project: combining active databases and timing constraints. *ACM Sigmod Record*, 17(1):51–70, 1988 (cited on p. 78).

[115] Narain H Gehani and HV Jagadish. Ode as an active database: constraints and triggers. In *Very Large Data Bases (VLDB)*. Volume 91, 1991, pages 327–336 (cited on p. 78).

[116] Michael Stonebraker, Eric N. Hanson, and Spyros Potamianos. The Postgres rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, 1988 (cited on p. 78).

[117] Matthew Morgenstern. *Active databases as a paradigm for enhanced computing environments*. Information Sciences Institute, 1983 (cited on p. 78).

[118] I. TIBCO. TIB/Rendezvous white paper. *Palo alto, california*, 1999 (cited on p. 79).

[119] B. Fitzpatrick, B. Slatkin, and M. Atkins. Pubsubhubbub protocol. 2010. URL: `http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html` (visited on 09/13/2013) (cited on p. 79).

[120] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001 (cited on p. 79).

[121] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei. A scalable and elastic publish/-subscribe service. In *Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pages 1254–1265 (cited on p. 79).

[122] 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. 80, 126, 194, 220).

[123] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. Volume 1. springer Heidelberg, 2006 (cited on p. 81).

[124] M. Cai, M. Frank, B. Yan, and R. MacGregor. A subscribable peer-to-peer RDF repository for distributed metadata management. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2(2):109–130, 2004 (cited on p. 81).

[125] D. Ranger and J.F. Cloutier. Scalable peer-to-peer RDF query algorithm. In *Web Information Systems Engineering (WISE)*. Springer, 2005, pages 266–274 (cited on p. 82).

[126] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Continuous RDF query processing over DHTs. In, *The Semantic Web - ISWC*, pages 324–339. Springer, 2007 (cited on pp. 82, 90).

[127] Y. Shvartzshnaider, M. Ott, and D. Levy. Publish/subscribe on top of DHT using RETE algorithm. *Future Internet Symposium (FIS)*:20–29, 2010 (cited on p. 82).

[128] Charles L Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982 (cited on p. 82).

[129] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Challenges for distributed event services: scalability vs expressiveness. *Proceedings of Engineering Distributed Objects*:72–78, 1999 (cited on p. 84).

[130] Laurent Pellegrino, Françoise Baude, and Iyad Alshabani. Towards a scalable cloud-based RDF storage offering a pub/sub query service. In *International conference on Cloud Computing, GRIDs, and Virtualization*, 2012, pages 243–246 (cited on p. 84).

[131] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the international conference on Software Engineering*. IEEE, 1998, pages 261–270 (cited on p. 85).

[132] Christoph Liebig, Mariano Cilia, and Alejandro Buchmann. Event composition in time-dependent distributed systems. In *Proceedings of the international conference on Cooperative Information Systems*. IEEE, 1999, pages 70–78 (cited on p. 86).

[133] Peter R Pietzuch, Brian Shand, and Jean Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1):44–55, 2004 (cited on p. 86).

[134] Masoud Mansouri-Samani and Morris Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96, 1997 (cited on p. 89).

[135] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. Performance analysis of exponential backoff. *IEEE Transactions on Networking*, 13(2):343–355, 2005 (cited on p. 95).

[136] 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 pp. 123, 129).

[137] 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. 124).

[138] Michael David Mitzenmacher. The power of two choices in randomized load balancing. PhD thesis. University of California, 1996 (cited on p. 124).

[139] Martin Raab and Angelika Steger. Balls into bins - A simple and tight analysis. In, *Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998 (cited on p. 124).

[140] Djelloul Boukhelef and Hiroyuki Kitagawa. Dynamic load balancing in RCAN content addressable network. In *Proceedings of the international conference on Ubiquitous Information Management and Communication*. ACM, 2009, pages 98–106 (cited on p. 127).

[141] Djelloul Boukhelef and Hiroyuki Kitagawa. Multi-ring infrastructure for content addressable networks. In, *On the Move to Meaningful Internet Systems*, pages 193–211. Springer, 2008 (cited on p. 127).

[142] Antonios Daskos, Shahram Ghandeharizadeh, and Xinghua An. PePeR: a distributed range addressing space for peer-to-peer systems. In, *Databases, Information Systems, and Peer-to-Peer Computing*, pages 200–218. Springer, 2004 (cited on p. 128).

[143] 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. 129).

[144] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic load balancing in distributed hash tables. In, *Peer-to-Peer Systems IV*, pages 217–225. Springer, 2005 (cited on p. 129).

[145] Quang Hieu Vu, Beng Chin Ooi, Martin Rinard, and Kian-Lee Tan. Histogram-based global load balancing in structured peer-to-peer systems. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):595–608, 2009 (cited on pp. 129, 140).

[146] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the symposium on Foundations of Computer Science*. IEEE, 2003, pages 482–491 (cited on p. 136).

[147] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks: algorithms and evaluation. *Performance Evaluation*, 63(3):241–263, 2006 (cited on p. 140).

[148] 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. 140).

[149] Elisabeth Freeman. *Head first design patterns*. O'Reilly Media, Inc., 2004 (cited on p. 149).

[150] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. *Acm sigplan notices*, 43(10):563–582, 2008 (cited on p. 152).

[151] Laurent Baduel, Françoise Baude, and Denis Caromel. Asynchronous typed object groups for grid programming. *International Journal of Parallel Programming*, 35(6):573–614, 2007 (cited on p. 154).

[152] Brian McBride. Jena: implementing the RDF model and syntax specification. In *SemWeb*, 2001 (cited on p. 165).

[153] Max Voelkel. Writing the semantic web with Java. In *Cdh seminar galway*, 2005 (cited on p. 165).

[154] Steve Graham, David Hull, and Bryan Murray. Web services notification. *OASIS Standard*, 2006. URL: `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn` (visited on 12/07/2013) (cited on p. 167).

[155] Mikko Tommila. A high performance arbitrary precision arithmetic package. 2001. URL: `http://www.apfloat.org/apfloat_java` (visited on 12/01/2013) (cited on p. 169).

[156] Alexandre Bourdin. Contribution au développement de la plateforme distribuée PLAY. Apprenticeship Report. 2012 (cited on p. 172).

[157] Ludovic Henrio and Justine Rochas. Declarative Scheduling for Active Objects. In *Symposium On Applied Computing*. Sung Y. Shin, editor. ACM Special Interest Group on Applied Computing. ACM, pages 1–6. URL: `http://hal.inria.fr/hal-00916293` (cited on p. 179).

[158] Iyad Alshabani, Alexandre Bourdin, Philippe Gibert, Christophe Hamerling, Matthieu Lauras, Laurent Pellegrino, Roland Stuehmer, and Yiannis Verginadis. D5.2.1 Assessment of the PLAY integrated platform. Project Deliverable PLAY. 2012. URL: `http://play-project.eu/documents/summary/3/205` (visited on 12/10/2013) (cited on pp. 189, 215).

[159] Yiannis Verginadis, Yoannis Patiniotakis, Nikkos Papageorgiou, Roland Stuehmer, and Iyad Alshabani. D5.2.2 Assessment of the PLAY integrated platform v2. Project Deliverable PLAY. 2013. URL: `http://play-project.eu/documents/summary/3/245` (visited on 12/10/2013) (cited on pp. 189, 215).

[160] Iyad Alshabani, Bastien Sauvan, Roland Stuehmer, and Thomas Morsellino. D2.5.2 PLAY federated middleware specification and implementation v2. Project Deliverable PLAY. 2013. URL: `http://play-project.eu/documents/summary/3/239` (visited on 12/10/2013) (cited on pp. 189, 216).

[161] Nicolas Salatgé. D1.2.1 Overall framework model. Project Deliverable SocEDA. 2013. URL: `http://goo.gl/NRhuXB` (visited on 12/10/2013) (cited on pp. 189, 216).

[162] LIG and I3S. D2.3.1 EventCloud: state-of-the-art and requirements. Project Deliverable SocEDA. 2013. URL: `http://goo.gl/Y8LfQL` (visited on 12/10/2013) (cited on pp. 189, 216).

[163] LIG and Liris. D2.3.2 Federated middleware specification and implementation v1. Project Deliverable SocEDA. 2013. URL: `http://goo.gl/oNL1tc` (visited on 12/10/2013) (cited on pp. 189, 216).

[164] Javier D Fernandez, Miguel A Martinez-Prieto, Claudio Gutierrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 2013 (cited on pp. 193, 219).

[165] Gero Mühl. Large-scale content-based publish-subscribe systems. PhD thesis. Darmstadt University of Technology, 2002. URL: `http://tuprints.ulb.tu-darmstadt.de/274/1/dissFinal.pdf` (visited on 01/08/2014) (cited on pp. 194, 219).

[166] Peter Triantafillou and Andreas Economides. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2004, pages 562–571 (cited on pp. 194, 219).

[167] Guoli Li, Shuang Hou, and H-A Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2005, pages 447–457 (cited on pp. 194, 219).

[168] KR Jayaram and Patrick Eugster. Split and subsume: subscription normalization for effective content-based messaging. In *International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2011, pages 824–835 (cited on pp. 194, 219).

[169] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In, *Databases, Information Systems, and Peer-to-Peer Computing*, pages 74–85. Springer, 2007 (cited on pp. 194, 220).

[170] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS reasoning and query answering on top of DHTs. In, *The Semantic Web - ISWC*, pages 499–516. Springer, 2008 (cited on pp. 195, 220).

[171] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank Van Harmelen. Scalable distributed reasoning using MapReduce. In, *The Semantic Web - ISWC*, pages 634–649. Springer, 2009 (cited on pp. 195, 220).

[172] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: distributed reasoning over large-scale semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305–316, 2009 (cited on pp. 195, 220).

[173] Trong-Tuan Vu and Fabrice Huet. A lightweight continuous jobs mechanism for mapreduce frameworks. In *International symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013, pages 269–276 (cited on pp. 195, 221).

[174] 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 01/09/2014) (cited on pp. 195, 221).

List of Acronyms

ADL Architecture Description Language 34
AO Active Object 27
API Application Programming Interface 10
AST Abstract Syntax Tree 183
BGP Basic Graph Pattern 20
BSBM Berlin SPARQL Benchmark 70
CAN Content Addressable Network vi
CC Cloud Computing 12
CE Compound Event 85
CEP Complex Event Processing 6
CJK Chinese, Japanese, and Korean 58
CPU Central Processing Unit 8
CSBV Continuous Spread By Value 90
CSMA Chained Semantic Matching Algorithm 90
DBMS Database Management System 78
DFS Distributed File System 53
DHT Distributed Hash Table 10
DNS Domain Name System 154
EC EventCloud 147
ECA Event-Condition-Action 79