Modular P2P-Based Approach for RDF Data Storage and Retrieval

Imen Filali, Laurent Pellegrino, Francesco Bongiovanni, Fabrice Huet and Françoise Baude
INRIA-I3S-CNRS, University of Nice-Sophia Antipolis
2004 route des lucioles, Sophia Antipolis, France
imen.filali@inria.fr, laurent.pellegrino@inria.fr, francesco.bongiovanni@inria.fr
fabrice.huet@inria.fr, francoise.baude@inria.fr

***Abstract*—One of the key elements of the Semantic Web is the Resource Description Framework (RDF). Efficient storage and retrieval of RDF data in large scale settings is still challenging and existing solutions are monolithic and thus not very flexible from a software engineering point of view. In this paper, we propose a modular system, based on the scalable Content-Addressable Network (CAN), which gives the possibility to store and retrieve RDF data in large scale settings. We identified and isolated key components forming such system in our design architecture. We have evaluated our system using the Grid'5000 testbed over 300 peers on 75 machines and the outcome of these micro-benchmarks show interesting results in terms of scalability and concurrent queries.**

***Keywords*- Semantic Web; Peer-to-Peer (P2P); Resource Description Framework (RDF); RDF data indexing; RDF query processing**

I. Introduction

The Semantic Web [1] promises to deliver a new experience of the Web through the usage of more structurally complex data based on the Resource Description Framework (RDF) data model [2]. Realising this vision in large scale settings will be hardly feasible without proper and scalable infrastructures such as the ones proposed by the Peer-to-Peer (P2P) community in the last decade. More specifically, Structured Overlay Networks (SONs) such as CAN (Content Addressable Network) [3] and Chord [4] have proved to be an efficient and scalable solution for data storage and retrieval in large scale distributed environments [5], [6]. These overlays, which offer a practical Distributed Hash Table (DHT) abstraction, use a variant of consistent hashing [7] for assigning keys to nodes. Consistent hashing distributes the keys uniformly among all nodes, which provides a lookup performance of $O(\log N)$ where N is the total number of nodes in the network. However, such protocols can not handle more advanced queries such as partial keywords, wildcards, range queries, etc. because consistent hashing is not order-preserving; it randomly distributes lexicographically adjacent keys among all nodes. More advanced SONs such as P-Grid [8] and PHT [9] introduced the capability to handle more complex queries (e.g., range or prefix queries) but are still limited regarding the expressiveness of the queries they support.

The need to specifically manage a large amount of RDF data has triggered the concept of *RDF store*, which can be seen as a kind of "database" allowing to query RDF data using advanced query languages such as SPARQL [10]. The first generation for RDF data storage systems has spawned centralized RDF repositories such as RDFStore [11], Jena [12] and RDFDB [13]. Although these RDF stores are simple in their design, they suffer from the traditional limitations of centralized systems such as single point of failure, performance bottlenecks, etc. The Semantic Web community can benefit from the research carried out in P2P systems to overcome these issues. As a result, the combination of concepts provided by the Semantic Web and P2P together with efficient data management mechanisms seems to be a good basis to build scalable distributed RDF storage infrastructure. SPARQL is a very expressive language and supporting it in a distributed fashion is challenging. Various solutions based on P2P solutions have been proposed to process RDF data in a distributed way [14]–[16], but their architectures are rather complex and lack flexibility.

To meet the storage and querying requirements of large scale RDF stores, we revisit, in this paper, a distributed infrastructure that brings together RDF data processing and structured P2P concepts while keeping simplicity, reusability and flexibility in mind. The proposed architecture, based on a modified version of CAN [3], does not rely on consistent hashing. We chose to store the data in a lexicographical order in a three dimensional CAN which *(i)* eases RDF query processing, *(ii)* reflects the nature of RDF triples *(iii)* retains the benefits of deterministic routing [17].

The contributions of this paper are *(i)* the design of a fully decentralized P2P infrastructure for RDF data storage and retrieval, based on three dimensional CAN overlay, written in Java with the ProActive [18] middleware, *(ii)* the implementation of the proposed design with clear separation between the sub-components of the whole API (e.g., storage component, query processing, etc.). This modular architecture means it is possible to substitute sub-components (e.g., using another local RDF store). Finally, *(iii)* the evaluation of the proposed solution through micro-benchmarks carried out on the Grid'5000 test bed.

The remainder of the paper is organized as follows. In Section II, we give an overview of the related work for RDF data storage and retrieval in P2P systems. In Section III, we

introduce the proposed distributed infrastructure for RDF data storage and retrieval and present our data indexing and query processing mechanisms. The experimental evaluation of our approach is reported in Section IV. Finally, Section V concludes the paper and points out future research directions.

II. Related Work

Many P2P-based solutions have been proposed to build distributed RDF repositories [20]. Some of them are built on top of super-peer-based infrastructure as in Edutella [14]. In this approach, a set of nodes are selected to form the super-peer network. Each super peer is connected to a number of leaf nodes. Super-peer nodes manage local RDF repositories and are responsible for query processing. This approach is not scalable for two main reasons. First, the super-peer nodes are a single point of failure. Second, it uses the flooding-like search mechanism to route queries between super-peers.

By using DHTs, other systems, such as RDFPeers [15], address the scalability issue in the previous approach. RDFPeers is a distributed repository built on top of Multi-Attribute Addressable Network (MAAN) [24]. Each triple is indexed three times by hashing its subject, its predicate and its object. This approach supports the processing of atomic triple patterns as well as conjunctive patterns limited to the same variable in the subject (e.g., $(?\mathtt{s}, \mathtt{p}_1, \mathtt{o}_1) \wedge (?\mathtt{s}, \mathtt{p}_2, \mathtt{o}_2)$). The query processing algorithm intersects the candidate sets for the subject variable by routing them through the peers that hold the matching triples for each pattern.

From a topology point of view, the structure that comes closest to our approach is RDFCube [16], as it is also a three dimensional space of subject, predicate and object. However, RDFCube does not store any RDF triples. It is an indexation scheme of RDFPeers. RDFCube coordinate space is made of a set of cubes, having the same size, called *cells*. Each cell contains an *existence-flag*, labeled *e-flag*, indicating the presence (*e-flag=1*) or the absence (*e-flag=0*) of a triple in that cell. It is primarily used to reduce the network traffic for processing join queries over RDFPeers repository by narrowing down the number of candidate triples so to reduce the amount of data that has to be transferred among nodes. GridVine [19] is built on top of P-Grid [8] and uses a semantic overlay for managing and mapping data and meta-data schemas on top of the physical layer. GridVine reuses two primitives of P-Grid: $\mathtt{insert}(\mathtt{key}, \mathtt{value})$ and $\mathtt{retrieve}(\mathtt{key})$ for respectively data storage and retrieval. Triples are associated with three keys based on their subjects, objects and predicates. A lookup operation is performed by hashing the constant term(s) of the triple pattern. Once the key space is discovered, the query will be forwarded to peers responsible for that key space.
From the data indexing point of view, almost of the proposed approaches for RDF data storage and retrieval use hashing approaches to map data into the overlay. However, even if this indexing mechanism enables the efficient key-based lookup, resolving more complex queries such as conjunctive queries may led to an expensive query resolution process.

III. CAN-Based Distributed RDF Repository

The aim behind the P2P infrastructure proposed in this work is the RDF data storage and retrieval in a distributed environment.

At the architectural level, it is based on the original idea of CAN [3]. A CAN is a structured P2P network based on a d-dimensional Cartesian coordinate space labeled $\mathcal{D}$. This space is dynamically partitioned among all peers in the system such that each node is responsible for storing data in a *zone* of $\mathcal{D}$. To store the $(\mathtt{k}, \mathtt{v})$ pair (`insert` operation in Figure 1), the key `k` is deterministically mapped onto a point `i` in $\mathcal{D}$ and then the value `v` is stored by the node responsible for the zone comprising `i`. The lookup (`retrieve` operation in Figure 1) for the value corresponding to a key `k` is achieved by applying the same deterministic function on `k` to map it onto `i`. The query is iteratively routed from one peer to its adjacent neighbors, with closest zones'coordinates to the searched key, until it reaches the node responsible for that key.

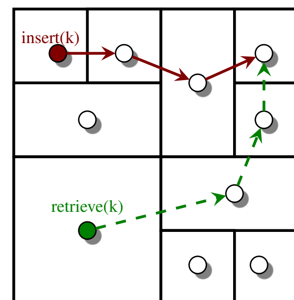


Figure 1. Routing in CAN: data storage ($\mathtt{insert}(\mathtt{k}, \mathtt{v})$) and retrieval ($\mathtt{retrieve}(\mathtt{k})$).

At the data representation level, data is presented in the RDF format [2]. RDF is a W3C standard aiming to improve the World Wide Web with machine processable semantic data. RDF provides a powerful abstract data model for structured knowledge representation and is used to describe semantic relationship among data. Statements about resources are in the form of <subject,predicate,object> expressions which are known as `triples` in the RDF terminology. The subject of a triple denotes the `resource` that the statement is about, the predicate denotes a `property` or a characteristic of the subject, and the object presents the `value` of the property. These triples, if connected together, form a directed graph where arcs are always directed from resources (subjects) to values (objects).

When designing an RDF data storage and retrieval, a set of key challenges have to be taken into account in order to come up with a scalable distributed RDF infrastructure. From the system scalability point of view, and unlike centralized solutions for massive RDF data storage

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-173-1

and retrieval which raise several issues (e.g., single point of failure, poor scalability), we argue that the use of a structured P2P overlay, at the architectural level, ensures the system's scalability. It also offers location transparency, that is, queries can be issued by any peer without any knowledge regarding the location of the stored data. Scalability needs to be achieved also at the query level by providing the ability to perform concurrent complex queries. As the data is expressed in RDF format, we use the SPARQL query language, which is another W3C recommendation [10], used to query RDF data. SPARQL queries could be in the form of:

- **atomic queries** are triples where the subject, the predicate and the object can either be variables or constant values. As an example, the query $\mathtt{q} = (\mathtt{s_i}, ?\mathtt{p}, ?\mathtt{o})$ looks, for a given subject $\mathtt{s_i}$, for all possible objects and predicates linked to $\mathtt{s_i}$. These kinds of queries are also called triple patterns.
- **conjunctive queries** are expressed as a conjunction of a set of atomic triple patterns (subqueries).
- **range queries** have specified ranges on variables. For instance, we consider the following query $\mathtt{q} = (< \mathtt{s} >< \mathtt{p} > ?\mathtt{o}\ \mathtt{FILTER}\ (\mathtt{v_1} \leq ?\mathtt{o} \leq \mathtt{v_2}))$ with a given subject $\mathtt{s}$ and a predicate $\mathtt{p}$. It looks for a set of objects, given by the variable $?\mathtt{o}$, such as $\mathtt{v_1} \leq \mathtt{o} \leq \mathtt{v_2}$.

As stated earlier, the intrinsic goal behind a distributed RDF storage is to search for data provided by various sources. As a first step towards this direction, we would like to guarantee that the data can be found as long as the source node responsible for that data is alive in the network. This can be guaranteed by using a structured overlay model for distributed RDF data storage and retrieval. In this work, our distributed RDF storage repository relies on a three dimensional coordinate space where each node is responsible for a contiguous zone of the data space and handles its local data store. In the following section, we detail the data storage and retrieval process.

A. RDF Data Organization

The RDF storage repository is implemented using a three dimensional CAN overlay with lexicographic order. The three dimensions of the CAN coordinate space represent respectively the subject, the predicate and the object of the stored RDF triple. From now, let us denote by $\mathtt{z_{smin}}$,$\mathtt{z_{smax}}$, $\mathtt{z_{pmin}}$,$\mathtt{z_{pmax}}$, $\mathtt{z_{omin}}$ and $\mathtt{z_{omax}}$, the minimum (min) and the maximum (max) borders of a peer's zone according to the subject axis ($\mathtt{z_{smin}}$, $\mathtt{z_{smax}}$), the predicate axis ($\mathtt{z_{pmin}}, \mathtt{z_{pmax}}$), and the object axis ($\mathtt{z_{omin}}, \mathtt{z_{omax}}$). We say that a triple $\mathtt{t} =< \mathtt{s}, \mathtt{p}, \mathtt{o} > \in \mathtt{z}$ only if $\mathtt{z_{smin}} \prec \mathtt{s} \prec \mathtt{z_{smax}}$, $\mathtt{z_{pmin}} \prec \mathtt{p} \prec \mathtt{z_{pmax}}$ and $\mathtt{z_{omin}} \prec \mathtt{o} \prec \mathtt{z_{omax}}$; where s, p, o present respectively the subject, the predicate, and the object of the triple t and z is a zone of the CAN overlay. Doing so, a triple represents a point in the CAN space without using hash functions.

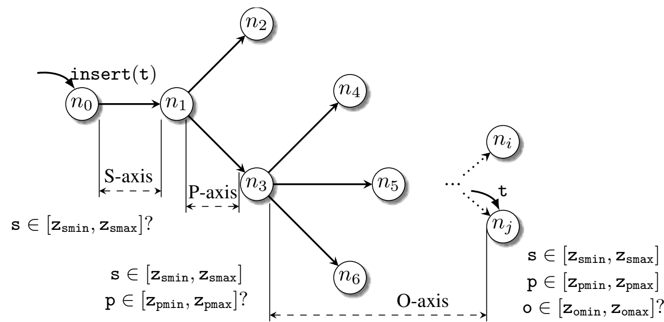


Figure 2. Insertion of RDF triples.

For better understanding, consider the example presented in Figure 2. Suppose that node n_0 receives an $\mathtt{insert(t)}$ request aiming to insert the RDF triple t in the network. Since no element of t belongs to the zone of n_0, and as s fits into the zone of n_1, n_0 routes the insert message to its neighbor n_1 according to subject axis ($\mathtt{S - axis}$). The same process will be performed by n_1, by means of the predicate axis ($\mathtt{P - axis}$) which in turn, will forward the message to its neighbor n_3. Once received, n_3 checks whether one of its neighbor is responsible for a zone such as o belongs to. Since the target peer is not found, the message will be forwarded at each step according to the object axis ($\mathtt{O - axis}$) to the neighbor with object coordinates which are closest to o. The idea behind this indexing mechanism is sketched in Algorithm 1.

Algorithm 1 Indexing algorithm

```
1:                                            ▷ Code for Peer P_i
2: upon event ⟨Insert | t⟩ from P_j
3:     if t ∉ Z_i then
4:         if s or p or o closer to one of my Neighbors'
   zones then
5:             send ⟨Insert | t⟩ to Neigh_dim
6:         end if
7:     end if
8: end event
```

This indexing approach has several advantages. First, it enables to process not only simple queries but also range queries. Using hashing functions in a DHT approach makes the management of such kind of queries expensive or even impossible. Moreover, in contrast to hashing mechanism that destroys the natural ordering of the stored information, the lexicographic order preserves the semantic information of the data so that it gives a form of clustering of triples sharing a common prefix. In other words, this approach allows that items with "close" values will be located in contiguous peers. As a result, range queries, for instance, can be resolved with a minimum number of hops. The routing

Copyright (c) IARIA, 2011.

process of an `insert` operation consists in finding the peer managing the zone where the triple falls to. Routing `query` messages is slightly more complex and will be explained later in this section.
A closer look reveals that one downside with this approach is that it is sensitive to the data distribution. RDF triples with common prefixes might be stored on the same peer, i.e., a node can become a `hot zone`. In the case where an element is common to many triples, such as a frequently occurring predicate (e.g., $< \texttt{rdf : type} >$), the triples can still be dispatched on to different peers, depending on the values of the other elements. However, when some elements share the same namespace or prefix, the probability that they end-up on a very small subset of all available peers is very high. To avoid this potential issue, we try to automatically remove namespaces or prefixes and only use the remaining part for indexing and routing. Some care has to be taken when doing this because if done too aggressively, we might lose the clustering mentioned earlier. Note that this issue also appears in other P2P implementations which rely on prefix-based indexing with order-preserving hash functions [19].

In the general case, there are other solutions that can be used to mitigate the impact of skewed data. First, one can limit the CAN space if some specific information is known about the data distribution. For instance, if it is known that all subjects will have a prefix falling in a small interval, then it is possible to instantiate the overlay with the specified interval, avoiding empty zones. Second, if at runtime some peers are overloaded, it is possible to force new peers to join zones managing the highest number of triples, hence lowering the load. Some more advanced techniques have been proposed to deal with imbalance such as duplicating data to underloaded neighbors or having peers manage different zones [3].

B. RDF Data Processing

From the data retrieval perspective, efficient data lookup is at the heart of P2P systems. Many systems, such as Chord, relies on consistent hashing to uniformly store ($\texttt{key}, \texttt{value}$) pairs over the key space. However, consistent hashing only supports key-based data retrieval and is not a good candidate to support range queries since adjacent keys are spread over all nodes as stated earlier. Therefore, efficient lookup mechanisms are needed to support not only simple atomic queries but also conjunctive and disjunctive range queries. Hereafter, we detail how the queries are supported by our routing process:

- **atomic queries** are routed on the $\texttt{subject} - \texttt{axis}$ of the CAN overlay looking for a match on the subject value $\texttt{s}_\texttt{i}$. Once a peer responsible for the specified value $\texttt{s}_\texttt{i}$ is found, it forwards the query through its neighbors in the dimension where peers are most likely to store corresponding triples based on their zones's coordinates
- **conjunctive queries** are decomposed into atomic queries and propagated accordingly.
- **range queries** are routed by first identifying the constant part(s) in the query. Then the lowest and the highest values are located by going over the corresponding axis. If all results are found locally, they are returned to the query initiator. Otherwise, the query is forwarded to neighbors that may contain other potential results.

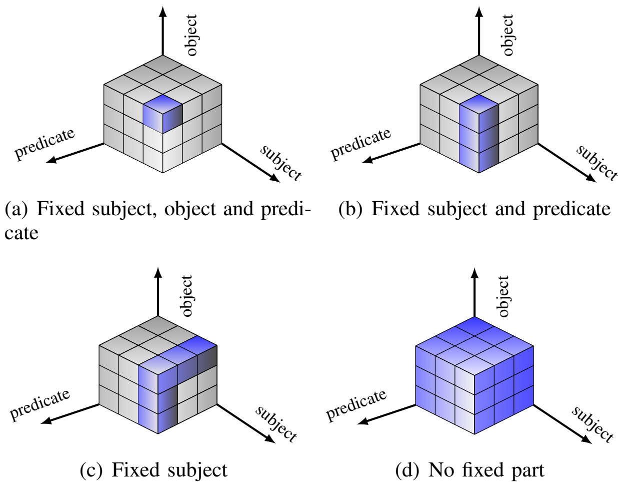
In order to route a query, a client sends it to a peer inside the overlay, the *query initiator*, as mentioned in Algorithm 2 at line 2. Once received, this query will be transformed, i.e., the initiator creates a message with additional information used for routing purposes (line 3), notably a *key* corresponding to the coordinates the message must be routed to. The next step consists of decomposing a complex query, a conjunctive query for instance, into atomic queries (line 5). Once we have these atomic queries, the peer sends messages, in parallel, to its neighbors accordingly, that is, if through them it can reach peers responsible for potential matches (lines 6 - 8). Whenever a peer has to propagate the message in different dimensions, it will de facto become a synchronization point for future results, that is, it waits for the results to come back and will merge the results before sending them to the client node (lines 15 - 21). In parallel of sending messages to its neighbors, the initiator will also check its local datastore in case it has potential matches for the query (line 10). Once neighbors receive a routing message (line 23), they will check their local datastore in case they can match the query (line 24) and return possible results to the initiator (line 26) otherwise they propagate the message to their neighbors accordingly (line 28). In order to ease the routing of the results, each message will embed the list of visited peers. This technique ensures that the forward path is the same as the backward path, avoiding potential issues related to NAT traversal, IP filtering, etc. that may happen in case we want to establish a direct connection to the initiator peer.

Figure 3 depicts various routing scenarios depending on the parts within the triple pattern. If subject, predicate and object are fixed, e.g., when performing an *add*, then the only peer which potentially holds matching results will be involved 3(a). In case the subject and the predicate are fixed, the message will have to traverse the object dimension in order to collect matching triples 3(b). When only the subject is fixed, the routed message will have to cross the object and predicate dimensions 3(c). Note that whenever a query with only variables is processed, our approach naively uses message flooding through each peer's neighbors. Hence, it may happen that a peer receives a message multiple times from different dimensions as shown in Figure 3(d). These duplicate messages will be ignored. Thanks to the way data is indexed and stored, queries are restricted to a specific subspace where candidate results are more likely to be

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-173-1

Algorithm 2 SPARQL queries routing algorithm

1: $\triangleright$ Code for the $Query\ Initiator$
2: **upon event** $\langle$Query $\mid Q\rangle$ **from** $client$
3: $RQ \leftarrow transformIntoRoutableQuery(Q)$
4: **if** RQ is a complex query **then**
5: List of $sub_q_i \leftarrow decomposeQuery(RQ)$
6: **for each** sub_q_i **do**
7: **send** $\langle$SubQuery $\mid sub_q_i\rangle$ **to** $Neigh_{dim}$
8: **end for each**
9: **end if**
10: **if** local RDFs matches Q **then**
11: $MergedRes \leftarrow MergedRes \cup \{local\ matched\ RDFs\}$
12: **end if**
13: **end event**
14:
15: **upon event** $\langle$SubQueryResults $\mid Res\rangle$ **from** $Neigh_{dim}$
16: $MergedRes \leftarrow MergedRes \cup \{Res\}$
17: $Pending_Sub_q \leftarrow Pending_Sub_q \setminus \{Neigh_{dim}\}$
18: **if** $Pending_Sub_q == \emptyset$ **then**
19: **send** $\langle$FinalRes $\mid MergedRes\rangle$ **to** $client$
20: **end if**
21: **end event**
22: $\triangleright$ Code for the Neighbors$_{dim}$
23: **upon event** $\langle$SubQuery $\mid sub_q_i\rangle$ **from** $Initiator$ or $Neigh_{dim}$
24: **if** local RDFs matches sub_q_i **then**
25: $Res \leftarrow matched\ RDFs$
26: **send** $\langle$SubQueryResults $\mid Res\rangle$ **to** $Initiator$
27: **else**
28: **send** $\langle$SubQuery $\mid sub_q_i\rangle$ **to** $Neigh_{dim}$
29: **end if**
30: **end event**



(a) Fixed subject, object and predicate (b) Fixed subject and predicate

(c) Fixed subject (d) No fixed part

Figure 3. Example of message scope depending on constant parts in the query.

found.

C. Modular Architecture

One of the goals when designing this distributed storage was to be able to easily change or modify some parts. A modular architecture is at the heart of the design, clearly separating the infrastructure (a CAN overlay), the query engine (using Jena [12]) and the storage system (a BigOWLIM [21] repository) as depicted in Figure 4. However, these elements do not work in isolation. Rather, they require frequent interactions. In this section, we will outline the different parts of our architecture, explaining their functions and showing their relations.

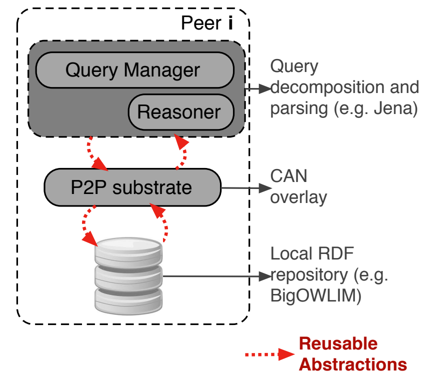


Figure 4. Modular architecture overview.

Query manager. Although the routing of the query is a P2P substrate's responsibility, part of the process requires the analysis of the query in order to extract atomic queries and their constant parts. This is performed using the Jena Semantic Web Framework [12] which provides dedicated operations. When a query returns data sets from multiple peers, the merge/join operation also relies on Jena. In order to experiment with the modularity aspect of our implementation, we have successfully swapped the query engine for Sesame [22] without impacting the other parts of the architecture.

P2P substrate. This "layer" is responsible for maintaining the CAN infrastructure, routing messages and accessing the local repository. The 3D CAN overlay is managed through an *Overlay* object which is responsible for maintaining a description of the zone managed by the current peer and an up-to-date list of its neighbors. Changing the number of dimensions of the CAN, e.g., to handle meta-data, requires providing a modified implementation of the *Overlay* object. To route a query, we first analyze it to determine its constant parts, if any, which will be used to direct it to the target peer. When there is not enough information to make a routing decision, it is broadcasted to the neighboring peers which will perform the same process.

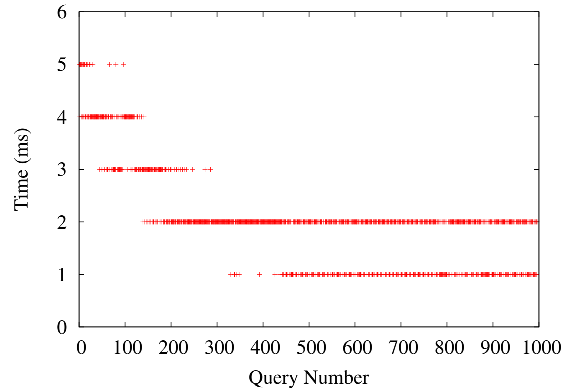
Local storage abstraction. The local storage is ultimately responsible for storing data and processing queries locally.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-173-1

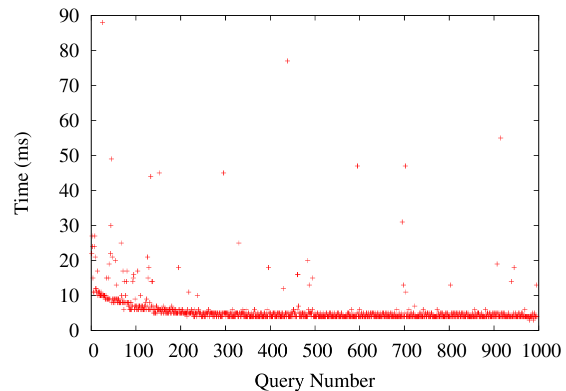
It is important for the P2P infrastructure to be independent from the storage implementation. All references are isolated through an abstraction layer whose role is to manage the differences between data structures and API between the P2P and the storage implementations. Some requests require the access to the local repository to read or write some information. Although this is rather straightforward, some care has to be taken regarding the commit of data to the storage. With some implementations like BigOWLIM, committing can take some time and thus should not be done after each write operation. The peer can implement a policy to only perform them when a threshold is reached (e.g., the time passed since last commit aka *commit interval*, number of write done, etc.) or when a read query has to be processed.

IV. Experimental Evaluation

In order to validate our framework, we have performed micro-benchmarks on an experimental testbed, Grid'5000. 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 BigOWLIM repositories per machine, each of them running in a separate Java Virtual Machine.



(a) on a single local peer



(b) on a remote peer

Figure 5. Insertion of 1000 statements with one peer.

A. 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 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 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.

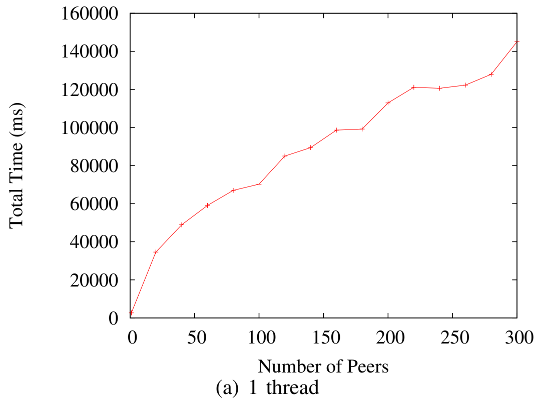
In a second experiment, the caller and the peer were put on separate hosts in order to measure the impact of the local network link on the performance. As shown in Figure 5(b), almost all add operations took less than 9 ms while less than 6.7% took more than 10 ms. The average duration for an add operation was 6 ms.

Multiple peer 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 6(a) and Figure 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 time is taken to add statements since more peers are likely to be visited before finding the target one. However, when performing the insertion concurrently, the total time is decreased but still depending on the number of peers. Depending on the various sizes of the zones of the global space and the first peer randomly chosen for triple's insertion, the performance can vary, as can be seen with the small downward spike on Figure 6(b) at around 80 peers. To measure the benefits of concurrent access, we have measured the time to add 1000 statements on a network of 300 peers while varying the number of threads from 1 to 50. Results in Figure 7 show a sharp drop of the total time, clearly highlighting the benefits of concurrent access.

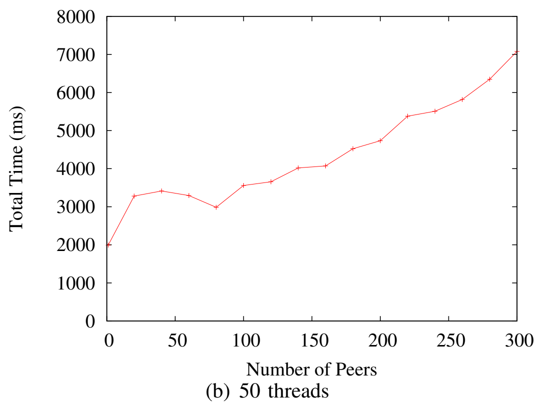
B. Queries using BSBM data

The *Berlin SPARQL Benchmark* (BSBM) [23] 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

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-173-1



(a) 1 thread



(b) 50 threads

Figure 6. Insertion of 1000 statements for a variable number of peers.

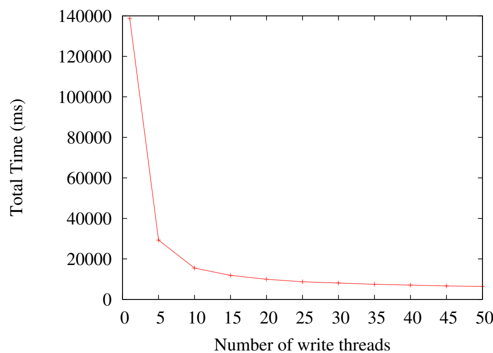


Figure 7. Evolution of the time for concurrent insertion with 300 peers.

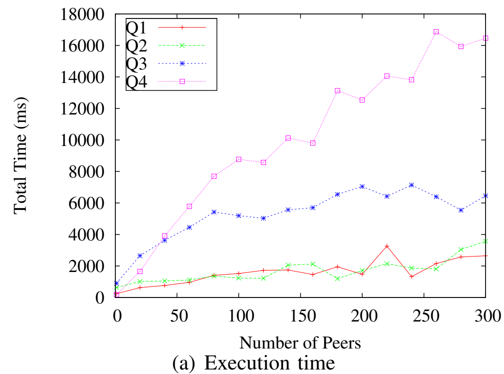
13320 Offers, 1 Rating Site with 339 Persons and 6660 Reviews. Out of this benchmark, we chose four queries to execute on our infrastructure:

- Q1 finds all the producers from Germany
- Q2 retrieves triples having "purl:Review" as object
- Q3 retrieves triples having "rdf:type" as predicate
- Q4 returns a graph where "bsbm-ins:ProductType1" instance appears

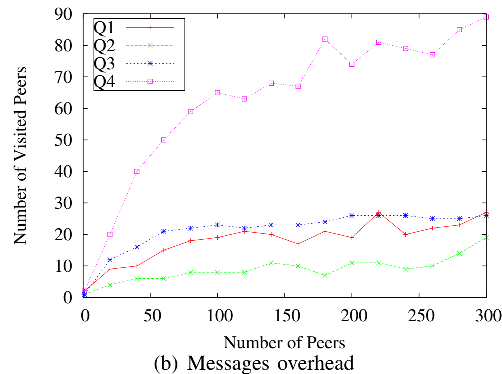
Q1 and Q4 are complex queries and will be decomposed into two subqueries. Hence, we expect a longer processing time for them. The number of matching triples for each query is as follows:

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

Figure 8 shows the execution time and the number of visited peers when 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 compared with other queries. Finally, Q4 generates two subqueries with two variables each, making it the request with the highest number of visited peers. In the network of 300 peers, the two subqueries have visited more than 85 peers.



(a) Execution time



(b) Messages overhead

Figure 8. Custom queries with BSBM dataset on various overlays.

Summary. Regarding the statement 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 the query processing phase strongly depends on the number of sub-queries, the payload carried between peers and the number of visited peers. While the payload depends on the complexity

of the query itself (conjunctive/atomic query, number of variables in a triple pattern, etc.), the number of visited peers depends not only on the structure of the overlay but also on the randomly chosen peer for initiating the query.

V. Conclusion

In this paper we have presented a distributed RDF storage based on a structured P2P infrastructure. RDF triples are mapped on a three 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 triples falling into it. We do not use hash functions, thus preserving the data locality. By removing constant parts such as prefixes from when indexing elements, we lessen bias naturally present in RDF data. The implementation has been designed with flexibility in mind. Our modular design and its implementation is abstracted away from the local storage and the query decomposer, thus they can swapped with other ones with minimal efforts. It also relies on standard tools and libraries for storing triples and processing SPARQL queries. We have validated our implemention with micro-benchmarks. Although basic operations like adding statements suffer from an overhead, the distributed nature of the infrastructure allows concurrent access. In essence, we trade performance for throughput. On a 75 nodes cluster, we have deployed an overlay of 300 peers. The time taken for query processing depends on the number of variable parts in the query and the size of the result set.

Future work. When queries have to be multicasted along different dimensions, the number of visited peers increases significantly, lowering the global performance. In this regard, we are currently working on an optimal broadcast algorithm, such as the one proposed in [25], which we adapt to CAN. This will allow us to decrease the number of redundant messages in case no constant parts are specified within the triple patterns of the query. Finally, we will investigate the impact of churn and node failures in our future experiments.

Code availability: The implementation mentioned in this paper is available at: http://code.google.com/p/event-cloud/.

Acknowledgment. The presented work is funded by the EU FP7 STREP project PLAY (http://www.play-project.eu) and French ANR project SocEDA (http://www.soceda.org)

References

[1] "W3C Semantic Web Activity," http://www.w3.org/2001/sw/, last accessed: July 2011.

[2] G. Klyne, J. Carroll, and B. McBride, "Resource description framework (RDF): Concepts and Abstract Syntax," *Changes*, 2004.

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content addressable network," in *Proceedings of SIGCOMM '01*, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of SIGCOMM '01*. New York, NY, USA: ACM, 2001, pp. 149–160.

[5] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 3540, 2010.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[7] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*, 1997, pp. 654–663.

[8] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: a Self-Organizing Structured P2P System," *ACM SIGMOD Record*, vol. 32, no. 3, p. 33, 2003.

[9] S. Ramabhadran, S. Ratnasamy, J. Hellerstein, and S. Shenker, "Prefix Hash Tree: An Indexing Data Structure Over Distributed Hash Tables," in *PODC*, 2004.

[10] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C Recommendation, January 2008, `http://www.w3.org/TR/rdf-sparql-query/`, last accessed: July 2011.

[11] "RDFStore," http://rdfstore.sourceforge.net/, last accessed: July 2011.

[12] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations," in *World Wide Web conference*. ACM, 2004, pp. 74–83.

[13] R.V.Guha, "rdfDB: An RDF Database," http://guha.com/rdfdb/, last accessed: July 2011.

[14] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch, "EDUTELLA: A P2P Networking Infrastructure Based on RDF," in *Proceedings of the 11 International World Wide Web Conference*, Honolulu, USA, May 2002.

[15] M. Cai and M. R. Frank, "RDFPeers: a Scalable Distributed RDF Repository Based on a Structured Peer-to-Peer Network," in *WWW*, 2004, pp. 650–657.

[16] A. Matono, S. Pahlevi, and I. Kojima, "RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores," *DBISP2P*, pp. 323–330, 2007.

[17] T. Schutt, F. Schintke, and A. Reinefeld, "Structured overlay without consistent hashing: Empirical results," in *Cluster Computing and the Grid Workshops, 2006. Sixth IEEE International Symposium on*, vol. 2, 2006, p. 8.

[18] "The ProActive middleware," http://proactive.inria.fr/, last accessed: July 2011.

[19] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. V. Pelt, "GridVine: Building Internet-Scale Semantic Overlay Networks," in *International Semantic Web Conference*, 2004.

[20] I. Filali, F. Bongiovanni, F. Huet and F. Baude, "A Survey of Structured P2P Systems for RDF Data Storage and Retrieval," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems III*, 2011, pp 20–55.

[21] D. M. Atanas Kiryakov, Damyan Ognyanov, "OWLIM : a Pragmatic Semantic Repository for OWL," 2005.

[22] A. K. Jeen Broekstra1 and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *The Semantic Web - ISWC 2002 In Proceedings of the first Int'l Semantic Web Conference*, 2002, pp. 54–68.

[23] C. Bizer and A. Schultz, "The berlin sparql benchmark," 2009.

[24] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A Multi-Attribute Addressable Network for Grid Information Services," in *Journal of Grid Computing*, vol. 2, 2003.

[25] S. El-Ansary, L. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured P2P networks," in *Peer-to-Peer Systems II*, 2003, pp. 304–314.