PAGE: A Distributed Infrastructure for Fostering RDF-Based Interoperability

Emanuele Della Valle, Andrea Turati, and Alessandro Ghioni

CEFRIEL - Politecnico of Milano, Via Fucini 2, 20133 Milano, Italy
dellavalle@cefriel.it, turati@cefriel.it, ghioni@cefriel.it

Abstract. This paper shows how to build a scalable, robust and efficient distributed Internet-scale RDF repository, that we name *PAGE* (**P**ut **A**nd **G**et **E**verywhere).

1 Motivation

In the recent years, the *RDF* (Resource Description Framework) *data model* is gaining momentum. Among other significant examples, take for istance the adoption of key players such as Adobe (i.e., the eXtensible Metadata Platform - XMP) and Oracle (i.e., the 10g version of the famous RDBMS promise to be the best RDF storage engine).

RDF enables the encoding of relational data over the Internet using *triples* usually denoted with `(spo)`, where `s` is the subject, `p` is the predicate and `o` the object. Complex structures can be easily encoded in a set of RDF triples. But, what about managing them? Centralized solutions may help in the short term, but if RDF should become the basis of the Semantic Web, some sort of distribution must be take into account. What if we could *put and get* triples without bothering of the underlying infrastructure because such infrastructure is the whole Internet?

Such a distributed and decentralized RDF repository is not yet available, but the peer-to-peer community has been studying the "**turn on, put() and get() out**" paradigm using Distributed Hash Tables (DHTs) for almost a decade. In this paper we describe *PAGE* (**P**ut **A**nd **G**et **E**verywhere), an optimized peer-to-peer infrastructure for distributed RDF storing and retrieval: an original approach that comes out from the convergence between the current works on DHT in the peer-to-peer community and on optimized index structures for querying RDF in the Semantic Web community named YARS [1].

Our work aims at advancing the state-of-the-art in decentralized RDF repositories finding a convergence between peer-to-peer (i.e., DHT) and Semantic Web researches. In particular it belongs to the class of solutions which avoid flooding approach, previously investigated by RDFPeers [2] and GridVine [3].

2 State-of-the-Art

Optimized RDF Storing and Retrieval. In a single RDF statement (also called triple) `(spc)` the subject `s` can be either an URI reference or a blank node

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 347–353, 2006.

(that is an abstract entity), the predicate `p` is an URI reference and the object `p` can be an URI reference, a blank node or a literal (e.g., a string). In order to group triples, a *context* is often associated to each triple. A triple and it's context is named quad and it is denoted with `(spoc)`, where `c` is the context.

In order to store and retrieve RDF triples, several storage systems have been developed. Many centralized RDF repositories have been implemented to support storing, indexing and querying RDF documents, such as RDFDB, Inkling, RDFStore, and Sesame. For this paper, we examine *YARS* [1], a recent solution that defines *an optimized index structure for fast retrieval of RDF* statements.

YARS index structure consists of two parts: the lexicon and the quad index. The former stores mappings from literals and URIs to internal object IDs (used for compactness), the latter stores the structure information (i.e. the quads).

Access patterns are strings that represents a set of RDF triples and are used as basic building blocks for more complex RDF query languages such as SPARQL. An access pattern is similar to a quad, in which at least one element is unspecified (and, for convention, is set to `?`). So, for example, `(s?oc)` means that we are interested at all quads having relations between the subject `s` and the object `o`, in the context `c` (it's equivalent to say that we want to get all predicates that have been stated together with `s`, `o` and `c`).

All possible access patterns are 16, but YARS's authors suggest to combine them in six indexes (SPOC, CP, OCS, POC, CSP, OS) because a single index can be used to cover more than a type of query (see [1] for details). For example, the SPOC index is used to process `(s???)`, but also `(sp??)` and `(spo?)`. An index is a structure that stores objects in a way that allows fast retrieval: YARS uses B+-trees, because they support range queries.

Distributed Hash Tables. In DHTs each resource is associated with a *key* which can be produced by hashing a significant information (e.g. the name) related to the resource. Nodes have identifiers in the same space of the keys. Each node is responsible for storing a range of keys and corresponding resources. The DHT nodes maintain a routing table in which IDs of several other nodes are stored. When a node performs a `get(key)` request, the lookup message is routed through the overlay network to the node responsible for the key.

Several DHTs (e.g., CAN, Chord, Tapestry) have been designed with different functioning mechanisms, but everyone with the same main concepts: the keyspace partitioning and an overlay network. The former refers to the technique used to assign certain sets of keys to different nodes and the latter refers to the structure of links used by nodes to communicate via message exchanging.

Most DHTs assign a single key at each node, called its ID (identifier). Then, keys are assigned to nodes according to a particular function that gives a notion of the distance between two keys: a node with ID x is the root for all the keys for which x is the closest ID, measured according to a distance function.

Given a key k, it is necessary to send a message to the root of k in order to retrieve the corresponding value: at each step, the message is forwarded to the

neighbour whose ID is closer to k, until there is no such neighbour, in which case we have arrived at the root node responsible for k. This process is called overlay routing and it can be done through the use of a routing table in every node. Indeed, each node maintains a set of links to other nodes (also called neighbours) such that for any key k, the node either is root for k or has a link to a node that is closer to k in terms of the keyspace distance. The more the number of hops in any route and the number of neighbours per node are low, the more efficient is routing. Among the various implementation available (see [4] for a comparison) we chose *Bamboo*, because it is suitable with the lookup expansion mechanism used in the retrieval process described in section 3.

3 Modifying DHT for Distributed RDF Storing and Retrieval

In order to design a distributed and decentralized system that allows efficient storage and retrieval of RDF statements, **we have created *PAGE*, a modified Bamboo DHT that implements YARS index structure**.

The rough idea is to store every quad in a YARS fashion in six indexes, using the `put` operation of key-value pairs available in DHTs, where values are quads and keys are special coding of the six IDs used in the six YARS indexes. This allows to exploit DHT routing method in order to implement a distributed version of B+-trees used in YARS.

Computing ID. Because we need a shared method for ID assigning, instead of using a centralized lexicon (like YARS), we followed DHT approach in using a hash function (indeed it can be used also in a distributed way with a statistical assurance to generate unique IDs).

The ID of a quad is built by concatenating four different parts: each part is the result of an hash function applied to the respective quad's element.

For example (see step 1 of figure 1), supposing that a server wants to put the quad "(m:Lennon m:sings m:Imagine m:70s)", the hashes of each quad parts could be: `1` for "m:Lennon", `F` for "m:sings", `A` for "m:Imagine" and `7` for "m:70s" (for readability, here we represent the 40 bytes long IDs with 5 digits).

Furthermore, in every ID it's necessary to encode the index that has been used to build the ID: to refer to SPOC index we use the digit `1`, `4` for CP index, `6` for OCS index, `9` for POC index, `C` for CSP index and `E` for OS index.

Storing Process. As proposed in YARS, for efficient lookups, every quad has to be stored six times (once for each index) respecting the sort of its four components imposed by the type of index. For simplicity, the six storage messages are wrapped in a *PAGE* method named **put**.

Taking the previous example as a guide (see step 2 and 3 of figure 1), in order to store the quad "`(1 F A 7)`" in the six indexes, we have to store the following six IDs: "`11FA7`" for the SPOC index, "`47F1A`" for the CP index, "`6A71F`" for the OCS index, "`9FA71`" for the POC index, "`C71FA`" for the CSP index, and "`EA1F7`" for the OS index.

When an owner wants to make a quad public it calls the `put` operation that constructs six messages (each containing an ID of the quad) and dispatches them. Every message is forwarded toward the node with the most similar (numerically closer) ID to that contained in the message according to Bamboo standard behaviour: this node is the root of that quad. The root stores the quad and its ID and informs nearby nodes to store a copy of that information (replicas).

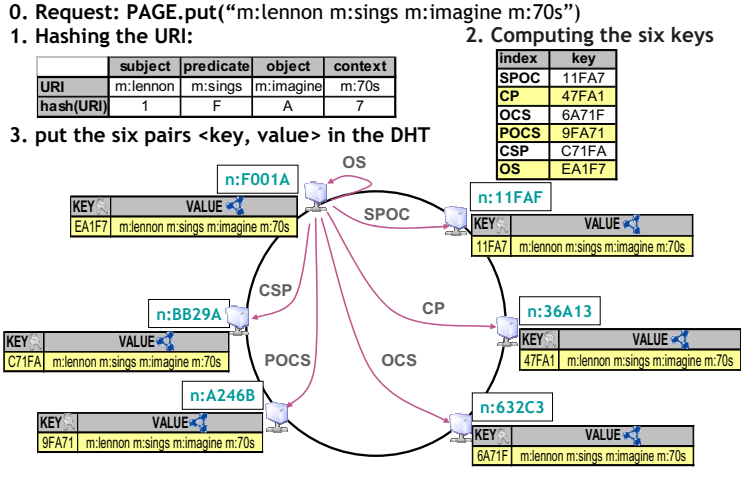

Fig. 1. Storing process

Retrieval Process. The *PAGE* **get** method wraps the query process: it takes an access pattern as input, computes the hashes, chooses the index, builds the access pattern ID and packages everything in a query message that is forwarded toward the corresponding root, which can answer with the asked quad.

Searching a single quad is not a very useful operation: if subject, predicate, object and context are already known, it means simply verify that that quad exists and it has been published. More interesting is asking quads of which some components are unknown (access pattern based query): `(sp?c)` finds all quads having specified subject, predicate and context, but having object unknown.

DHTs are designed mainly to perform efficient retrievals of individual values. The basic implementation of Bamboo does not manage queries about a range of values. Therefore, we have to modify a method that is able to perform single key lookups in order to build a method that allows lookups of a range of keys.

Queries have to be routed as messages, exactly as it happens for lookups of single quads, and thus we need to assign IDs to them.

At every possible query corresponds an access pattern. In order to assign an ID to a query it is sufficient to encode the corresponding access pattern by computing the hash of each its component (conventionally assigning a sequence of `0` to unspecified parts, pointed out with a `?`) and then reorder them respecting

the components' order imposed by the index that allows to answer to such query. Furthermore, a "*mask*" is associated to every query, likewise to what IP does in Internet where the netmasks are used for distinguishing some net addresses.

Since the zeros appear only at the end of every query ID, a mask is simply a flag that separates the known parts of the quad from those that we want to get (i.e. it points out the preceding position at that in which zeros begin). A mask is a number that specifies the position beyond which the quad parts is unspecified.

For instance (see figure 1), the access patter (m:Lennon m:sings ? ?), meaning all the songs performed by Lennon in any context, requires the SPOC index and it is converted to the ID `11F00/ 3`. For this query, the mask is `3` and means that the first 3 digits of the ID are relevant (the index and the first two quad parts are specified), so only the last two elements of the quad are not specified (i.e., the object and the context). The mask, along with the index code, allows for realizing what type of query has to be processed. The first digit of the ID says what index we have to use, but this information is not sufficient to correctly process a query: it is necessary to know the exact point beyond which we want all values, i.e. the query pattern.

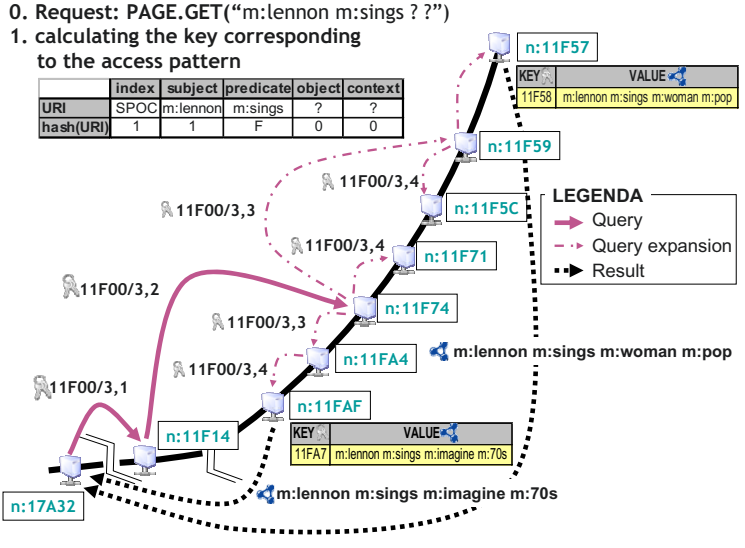

Fig. 2. Retrieval process

Along with the mask, another number is associated with the queries ID: "*hop*", a sort of reverse time-to-live, equals to the number of digits of the query ID already considered during routing process. When a query is introduced in the network its ID is accompanied by a specific mask and a hop set to 0. While the message goes through the network, the hop increases by 1 at each crossed node.

The routing of query messages has to exactly begin equally to the routing of lookups: every single node forwards the message to its neighbours that possesses an ID that has a longer matching prefix with the query ID, until the final part of the query ID (beginning from the digit following that pointed out by the mask) is reached. At that point, in fact, the message has to be sent to all the nodes that share the same prefix with the query ID (obviously excluding the sequence of zeros), because we are interested to any value that quads state in that part. So, every node forwards the message to several nodes that are in its routing table, increasing the hop so that the next node will know at which nodes it has to forward the message (it is like if the matching prefix were every time longer).

For example, suppose that a node with ID `17A32` wants to perform the query `11F00/3,0`: since it has the first digit in common with the query, it forwards the message `11F00/3,1` to a known node that has the first two digits in common with the query, whose ID is for instance `116D4`. The node `116D4` forwards message `11F00/3,2` to the node having the same first three digits (e.g. `11F74`).

Now, since the next digit to be matched is beyond the mask, the node `11F74` has to forward the message to all known nodes having the same prefix as query. Therefore, it sends the message `11F00/3,3` to all known nodes with ID in the range `11F00x-11FFx` and the message `11F00/3,4` to all known nodes whose ID is in the range `11F70-11F7F`: particularly, nodes `11F59` and `11FA4` receive the message `11F00/3,3` and proceed in the same way, while node `11F71` receives `11F00/3,4` and stop the propagation. Finally, all nodes involved in the lookup expansion send to the petitioner all quads that match the query prefix up to the mask.

4 Discussion and Conclusions

In this paper we have introduced *PAGE*, a distributed infrastructure that enables to *put* and *get* triples from everywhere, *simplifies the implementation.* PAGE *makes deploying* distributed Semantic Web applications *straight forward* because it avoids to implement from scratch all of the scalable routing, robustness, and management properties. Applications get all these properties at the cost of making available enough resources for processing the information it intends to exchange. Moreover, being PAGE self-organizing, each node is essentially independent from all other nodes and it only has to bother on making available enough resources to the infrastructure to make it grow incrementally. Therefore deploying an application just means either installing such application on pre-existing nodes, or adding nodes (with the application on top) to the infrastructure.

An advantage of our approach is the parallel execution of a query in the case of a lookup expansion: at the same time, several nodes retrieve a subset of query result. Because of results are returned when available, a clear drawback of PAGE is the impossibility of calculating the completeness of the result set: this aspect is a diffused problem of distributed systems that could be bypassed introducing a time threshold within which results are expected and beyond which results are rejected.

References

1. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: 3rd Latin American Web Congress, Buenos Aires - Argentina. (2005)
2. Cai, M., Frank, M.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: WWW '04, New York, NY, USA, ACM Press (2004) 650–657
3. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Pelt, T.V.: GridVine: Building Internet-Scale Semantic Overlay Networks. In: ISWC. (2004) 107–121
4. Li, J., Stribling, J., Morris, R., Kaashoek, M.F., Gil, T.M.: A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In: Proceedings of the 24th Infocom, Miami, FL (2005)