A Distributed Publish/Subscribe System for RDF Data

Laurent Pellegrino, Fabrice Huet, Françoise Baude, Amjad Alshabani

A Distributed Publish/Subscribe System for RDF Data

Laurent Pellegrino, Fabrice Huet, Françoise Baude, and Amjad Alshabani

INRIA-I3S-CNRS, University of Nice-Sophia Antipolis
2004 Route des Lucioles, Sophia Antipolis, France
firstname.lastname@inria.fr

Abstract. The pub/sub communication style is a prevalent messaging pattern for filtering information from distributed and large-scale network (e.g., from the real-time web, sensor networks, etc.) thanks to the decoupling between publishers and subscribers. At the same time, persisting the published information is a prerequisite for any further batch analytics on such big amount of data. As data can be heterogeneous, reliance on format from the semantic web such as RDF is unavoidable. In this paper we introduce two versions of a content-based pub/sub matching algorithm for RDF described events, working on an adapted version of the CAN structured P2P network designed to both store and disseminate RDF events. In contrary to existing pub/sub solutions based upon structured overlay networks that index semantic events several times due to the use of hash functions, we leverage the lexicographic order of the event elements. Thus, only subscriptions and not publications have to be duplicated, which is better given that in real settings, publications may occur more frequently than subscriptions. Furthermore, our system allows to publish events made of any number of elements and the subscription language leverages the SPARQL query language. The first algorithm we introduce initially derives from the ideas discussed by Liarou. et al. based upon rewriting continuous queries along matching RDF elements (CSBV) with the purpose to perform the matching between subscriptions and several RDF elements on multiple nodes. The experimental results discuss the applicability of the presented algorithms to some synthetic scenarios and identify, accordingly, which pub/sub matching algorithm is the more relevant.

1 Introduction

The advent of the Semantic Web by the precursor Tim Bernes-Lee incites available information on the World Wide Web to become more and more structured. Structured contents are possible thanks to powerful data models such as Resource Description Model (RDF) that makes knowledges machine-processable and machine-understandable. Many centralized solutions such as Jena [4], Sesame [1] or OWLIM [11] have been proposed the last years to store and retrieve RDF data. However, they all suffer from their inherent design that is not suitable to scale with the perpetual increase of the resources available on the Web. Some

decentralized approaches have been introduced to overcome this limitation. Most of them relies on Peer-to-Peer (P2P) networks that are recognized as a key communication model to build distributed and reliable applications at very large scale [10]. Usually, structured P2P protocols are provided with a standard abstraction called Distributed Hash Table (DHT) that offers a simple *put(key, value)* and *get(key)* API to store and fetch data. Even though such an abstraction is really well suited for manipulating key/value pairs, it does not support complex queries such as conjunctives and range queries that are at the core of SPARQL [17], the main query language for RDF data. Furthermore, the traditional query/response model is not designed for processing data streams.

Publish/subscribe systems are a natural extension of one-time queries where users formulate meaningful inquiries about their concern and wait for an answer. Unlike one-time queries that are synchronous, pub/sub systems assume that users register their needs through subscriptions also dubbed continuous-queries. As the name suggests, continuous-queries are resolved as soon as incoming information or events match subscribers' interests. Events are published to a brokering network in charge of performing the matching between the publications and the subscriptions that have been registered. Once an event is matched, a notification is triggered to the subscriber(s). Thus, users are kept updated efficiently and gradually

In this paper we focus mainly on the synergy between RDF-based P2P systems and the pub/sub messaging paradigm. Section 2 gives an overview of the existing works regarding this context. In Section 3 we introduce our data and subscription model along with our system properties. Section 4 enters into the details of the matching algorithms we have developped. Then, we bring out some information about the implementation before to introduce and discuss the experimental results in Section 5. Finally, Section 6 concludes.

2 Related Works

The last two decades, the flexibility, modularity and responsiveness of pub/sub led to the emergence of several solutions. These systems are classified into *topic-based* or *content-based* categories according to their expressivity. Tibco [23] and Pubsubhubbub [7] are representatives of this former category that provides limited filtering capabilities. Most prominent solutions regarding the latter category are certainly Siena [5] and Hermes [16]. 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 [20], a structured P2P protocol named PAN. Subscriptions and publications are sent to a rendez-vous node and notifications are forwarded by using reverse paths. More recently, BlueDove [13] propose to match publications with subscriptions atop a modified version of Cassandra [12] 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. The closest system to our is certainly Meghdoot [9]. The authors leverage the CAN [19] logical topology as we do. However, event types, domain (e.g, from 1 to 100 for event type integer) and the maximum number of attributes per event should be defined at startup. Moreover, the initial CAN configuration strongly depends on this last parameter.

RDFPeers [3] is a distributed RDF repository where peers are self-organized into a Multi-Attribute Addressable Network (MAAN) [2]. MAAN extends Chord [22] 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 subscriptions 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 [18], Ranger et al. introduce an information sharing platform for disseminating RDF activities. Their solution relies on the Scribe [6] system that offers a topic-based pub/sub 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 [14] 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 matching algorithm we introduce in the next sections derives from this idea.

Recently, Shvartzshnaider et al. proposed in [21] to combine AI and Peer-to-Peer research approaches for building a pub/sub system that supports publication of arbitrary tuples and subscriptions with standing graph queries. Their idea consists in applying Rete [8] 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, and also their results are cached for future reuse. Thus, only the changed data are matched against subscriptions. Publications and subscriptions are indexed similarly to RDFPeers 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.

3 EventCloud Design

In this section we give a description of the data and subscription model used by our system, dubbed EventCloud. We explain how events and more specifically how RDF data, along with subscriptions, are indexed in a CAN network.

3.1 Data and subscription model

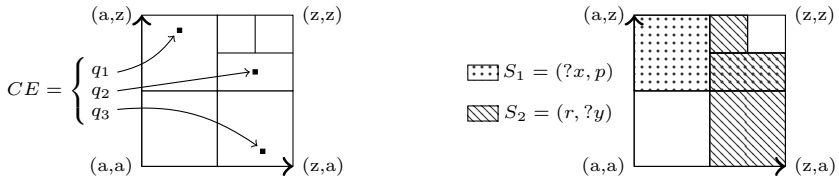
Our data and subscription model follows the approach taken in [15] to allow users to formulate queries and subscriptions but also to insert and publish information with the same models, that is respectively RDF and SPARQL.

Events The data are expressed in the RDF model using 4-tuples (quadruples) whose elements are named RDF terms. In our system an RDF term may be either an IRI or a Literal value. Elements generated at the same time by a given source form a *Compound Event* (*CE*), as defined by (1b). Each CE is made of a list of quadruples and all 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 twofold. It is used to identify the event source, the event itself and also to offer the possibility to link together several quadruples for emulating, yet unbounded, multi-attribute values like in traditional pub/sub systems.

$$q = (g, s, p, o) \mid g, s, p, o \in RDFTerm \tag{1a}$$

$$CE = (q_1, ..., q_i, ..., q_n) \mid q_i = (g, s_i, p_i, o_i) \tag{1b}$$

The EventCloud is based on a four dimensional CAN overlay that uses the lexicographic order for routing requests. The four dimensions of the CAN coordinate space are mapped respectively to the graph, the subject, the predicate and the object of any RDF 4-tuple that is indexed. One benefit of this approach is that a quadruple represents a point in the four dimensional Cartesian space. Hence a quadruple will only be stored by a single peer of the overlay. This indexing approach has several advantages. First, it supports range queries (looking for values in a specified range) efficiently. Second, the lexicographic order preserves the data semantics so that is gives a form of clustering of quadruples sharing a common prefix. In contrast, hash-based approaches destroy the natural ordering of information and make the management of complex queries difficult and expensive. The Figure 1 shows how CEs and subscriptions are mapped to a CAN network.



(a) Distribution of a Compound Event with 3 quadruples.

(b) Distribution of 2 subscriptions, overlapping on a peer.

Fig. 1. Example data and subscription distribution on a 2D CAN.

Subscriptions A subscription is content-based and formulated using a subset of SPARQL. It is basically a list of atomic queries called *sub-subscriptions* or *SS* with possibly a `FILTER` clause. 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 matched by a subset of the quadruples contained by a CE.

```
1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2  SELECT ?id ?name WHERE {
3      GRAPH ?g {
4          ?id foaf:name ?name .    // the point at the end of line
5          ?id foaf:age ?age        // stands for the and operator
6      }
7      FILTER (?age > 25)
8  }
```

Listing 1.1. Example of a SPARQL subscription.

For instance, the example depicted in Listing 1.1 states that all events that are related to an entity, with a name and whose age is greater than 25, have to be delivered to the subscriber.

To index the subscription in the overlay, we use a similar scheme than for the events. The subscription is transformed into a set of RDF quadruples. This set is made of the first sub-subscription and additional quadruples. The first one contains a unique identifier, the second one a timestamp and the last one the whole subscription. This set is then sent to the peers responsible for the fixed parts of the first sub-subscription. The subscription presented in the previous example is decomposed as shown in Listing 1.2 and S will be stored on the peers responsible for *?id foaf:name ?name*, i.e. those with a zone with value *foaf:name* on the predicate dimension, and any value on all others (graph, subject and object).

```
1  S = { (?g, ?id, foaf:name, ?name),
2        (id and timestamp), (subscription) }
```

Listing 1.2. Decomposition of a SPARQL subscription in RDF sets.

3.2 System properties

In addition to the data and subscription model, our framework also enforces a set of properties. If a *Compound Event* is added to the overlay, where there exist a matching subscription, then it will be delivered to the subscriber. Conversely, no *false positive* should be delivered. Finally, the causal ordering of publish and subscribe requests will be maintained for a given client (acting both as publisher and subscriber). If a subscription is issued before a matching compound event by the same client, then a notification should be issued.

4 Publish/Subscribe Algorithms

This section introduces two pub/sub 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) is optimized for the subscriptions.

4.1 CSMA

The general idea of CSMA, as inspired by Liarou et al., is to publish in parallel and perform the matching of all the sub-subscriptions sequentially. Indeed, all peers involved in a subscription will be organized in a chain-like fashion. Only the peers indexing the subscription, as described in Section 3.1 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.

Event Routing: A *Compound Event* upon entering the overlay is divided into simple events and each of them is published independently. There are two situations which can trigger the search for a subscription match.

Event Reception: When receiving an event, a peer checks whether there is a matching subscription or not. If there is one and the current event satisfies the *first* atomic query, then the subscription is rewritten into S'. The rewrite operation consists in replacing the variables of the atomic query with the event elements, basically striping down the subscription of the matched values. It also adds the unique identifier of the *Compound Event* so that the rewritten subscription can only match the remaining events. This new subscription is then sent into the overlay for re-indexing and potential matching.

Subscription Reception: When a peer receives a new subscription, it checks for events generated after the subscription but received before, using the timestamp value. This situation could occur because of events and subscription taking different path in the overlay, but more often because of the rewriting described earlier. Hence, this mechanism will allow for a rewritten subscription to match events from the same *Compound Event*, even if they were received earlier.

Finally, when the last matching is performed, the subscriber is notified about the corresponding graph value and can begin the reconstruction process.

Reconstruction: When notified of the graph value g of a matching *Compound Event*, a subscriber performs a reconstruction operation to retrieve the whole event. It synchronously queries the overlay with $(g, ?s, ?p, ?o)$ quadruple pattern to retrieve all events of the *Compound Event*. Since some events might still be routed to the correct peer in the overlay, we use a timeout based algorithm, i.e. if some events are still missing the subscriber waits a fixed duration before re-issuing a new request.

The main drawback of CSMA is that matching a subscription is essentially a sequential process with a complexity (number of steps) equals to the number of sub-subscriptions. Although any peer in the chain can receive a matching event, an ordering constraint is imposed by the subscription. Hence, we don't take advantage of the distributed nature of the subscription.

Furthermore, CSMA may suffer from duplicate notifications. This can happen when there is not a single peer at the end of the matching chain. For example, if the last sub-subscription, after rewriting, still contains variables, then it will be indexed on multiple peers and potentially trigger multiple notifications. Thus a filtering at subscriber side has to be performed during the reconstruction.

4.2 OSMA

To alleviate the previous issues, we propose a second algorithm, OSMA, which allows for parallel matching of subscriptions.

Routing: Instead of indexing only individual quadruples, we now index the whole *Compound Event* on each peer using each quadruple as a key.

Event Reception: When receiving the whole CE, the peer first stores only the quadruples which fall in its responsibility zone. Then, it looks for the subscriptions satisfied by the whole *Compound Event*.

Notification Triggering: Before notifying the subscriber of a match, some care has to be taken to avoid duplicate notifications. Indeed, potentially all peers storing the subscription and involved in the indexation of quadruples from the CE have now enough information to notify the match. To ensure only one peer sends the notification, we apply the following rule. A peer notifies a match *if and only if* it is responsible for the *first* of the matching events of the CE. For instance, if we have a $CE = (q_1, q_2, q_3)$, a subscription $S = (SS_1)$ and two peers P_1 and P_2 that receive respectively q_2 and q_3 and index both SS_1. If q_2 is the first quadruple from the CE that satisfies SS_1 on P_1, then only P_1 notifies the subscriber.

The main benefit of this algorithm is the expected low latency for subscribers. As soon as the CE reaches the peer responsible for the first matching event, the notification if 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. Also, note that this 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.

To summarize, the different properties of the two algorithms are presented in Table 1.

	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 1. Comparison of the two pub/sub algorithms.

5 Experiments

The latest version of our system dubbed EventCloud is publicly available[1] as an open source project under the AGPL license. From an implementation point of view each peer embeds Jena TDB instances for data and subscription storage.

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 are using 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.

In the first experiment we evaluate the effect of increasing the network size. For this purpose we place 1 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 2(a) depicts the average subscriber

[1] http://eventcloud.inria.fr

throughput[2], i.e. the throughput perceived on the subscriber when the network size it 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.

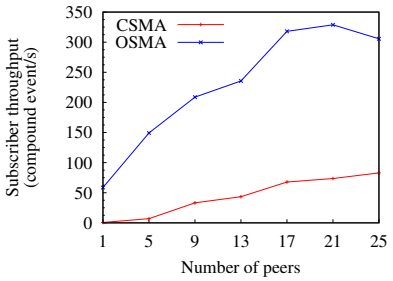
In a second experiment we evaluate the effect of varying the number of publications. Figure 2(b) shows that the throughput on the subscriber is constantly increasing with OSMA when the number of publications increases. This is because the overlay is not working at its full capacity when $x = 30 \times 10^3$ 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 insertions and the notifications. Owing to the reason that 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 with various number of subscriptions. The subscriptions are generated to match an equal number of *Compound Events*. Figure 2(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 Jena TDB. On the contrary, CSMA remains almost stable with a throughput that varies around 92 CEs per second. This effect is 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 different peers, thus, increasing the number of peers involved in the matching.

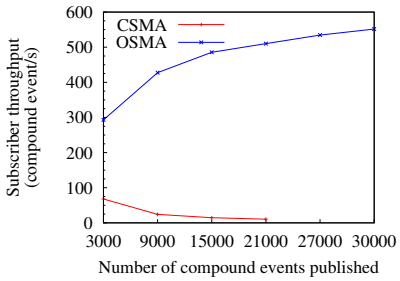
In a fourth experiment we test the effect of varying the number of peers when selective subscriptions are replaced by a subscription that accepts all events (c.f. Figure 2(d)). In such a situation, all peers index the subscription `SELECT ?g WHERE { GRAPH ?g { ?s ?p ?o }}`. As explained in Section 4.1, 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.

Finally, the time taken to store different number of publications with no subscription registered on peers is depicted on Figure 2(e). This criteria is directly related to the bandwidth consumption since no matching is performed. Indeed,

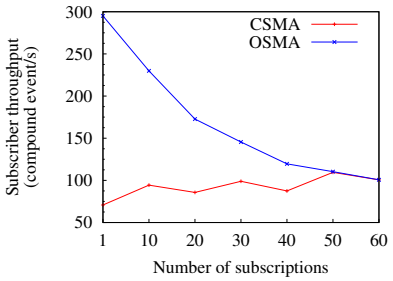
[2] Mathematically speaking it is defined as the number of matching publications divided by the time elapsed between the first notification and the last awaited notification received by the subscriber.



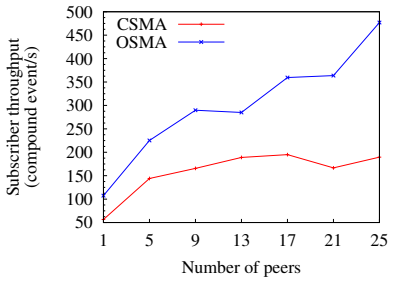
(a) Impact of overlay size. 3000 CEs published, one subscription of $k = 5$ quadruple patterns.



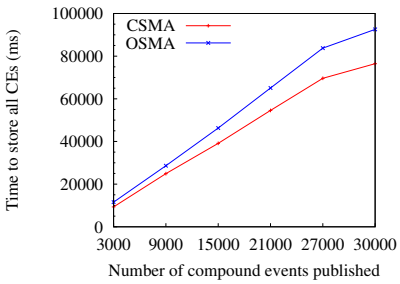
(b) Impact of the number of publications. 25 peers, one subscription ($k = 5$).



(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.

Fig. 2. Performance comparison of CSMA and OSMA.

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 responsibles to store quadruples. Thus, it will require more bandwidth than CSMA.

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.

6 Conclusion

In this paper we have introduced a pub/sub framework based on the RDF data model and SPARQL filter model. Subscribers can express their interests using 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. 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 the 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. Both algorithms have been experimentally tested for throughput and scalability.

Acknowledgments

This work was in part supported by the EU FP7 STREP project PLAY and French ANR project SocEDA. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed (see https://www.grid5000.fr). The authors wish to thank Bastien Sauvan, Iyad Alshabani, Justine Rochas and Maeva Antoine for their help with the implementation.

References

1. Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. The Semantic Web—ISWC 2002 pp. 54–68 (2002)
2. Cai, M., Frank, M., Chen, J., Szekely, P.: Maan: A multi-attribute addressable network for grid information services. Journal of Grid Computing 2(1), 3–14 (2004)
3. Cai, M., Frank, M., Yan, B., MacGregor, R.: 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)
4. Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters. pp. 74–83. ACM (2004)

5. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems (TOCS) 19(3), 332–383 (2001)
6. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.: Scribe: A large-scale and decentralized application-level multicast infrastructure. Selected Areas in Communications, IEEE Journal on 20(8), 1489–1499 (2002)
7. Fitzpatrick, B., Slatkin, B., Atkins, M.: Pubsubhubbub protocol (2010), http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html
8. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (1982), http://dx.doi.org/10.1016/0004-3702(82)90020-0
9. Gupta, A., Sahin, O., Agrawal, D., Abbadi, A.: Meghdoot: content-based publish/subscribe over p2p networks. In: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware. pp. 254–273 (2004)
10. Jelasity, M., Kermarrec, A.: Ordered slicing of very large-scale overlay networks. In: Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on. pp. 117–124. IEEE (2006)
11. Kiryakov, A., Ognyanov, D., Manov, D.: Owlim–a pragmatic semantic repository for owl. In: Web Information Systems Engineering–WISE 2005 Workshops. pp. 182–192. Springer (2005)
12. Lakshman, A., Malik, P.: Cassandra—a decentralized structured storage system. Operating systems review 44(2), 35 (2010)
13. Li, M., Ye, F., Kim, M., Chen, H., Lei, H.: A scalable and elastic publish/subscribe service. In: Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. pp. 1254–1265. IEEE (2011)
14. Liarou, E., Idreos, S., Koubarakis, M.: Continuous rdf query processing over dhts. In: Proceedings of the 6th international semantic web conference. pp. 324–339. Springer-Verlag (2007)
15. Pellegrino, L., Baude, F., Alshabani, I.: Towards a scalable cloud-based rdf storage offering a pub/sub query service. In: The Third International Conference on Cloud Computing, GRIDs, and Virtualization. pp. 243–246 (2012)
16. Pietzuch, P., Bacon, J.: Hermes: A distributed event-based middleware architecture. In: Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on. pp. 611–618. IEEE (2002)
17. Prud'Hommeaux, E., Seaborne, A., et al.: Sparql query language for rdf. W3C recommendation 15 (2008)
18. Ranger, D., Cloutier, J.: Scalable peer-to-peer rdf query algorithm. In: Web Information Systems Engineering–WISE 2005 Workshops. pp. 266–274. Springer (2005)
19. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. ACM SIGCOMM Computer Communication Review 31(4), 160–172 (2001)
20. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware 2001. pp. 329–350. Springer (2001)
21. Shvartzshnaider, Y., Ott, M., Levy, D.: Publish/subscribe on top of dht using rete algorithm. Future Internet-FIS 2010 pp. 20–29 (2010)
22. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review 31(4), 149–160 (2001)
23. TIBCO, I.: Tib/rendezvous white paper. Palo Alto, California (1999)