iBroker: An Intelligent Broker for Ontology Based Publish/Subscribe Systems

Myung-Jae Park [1] and Chin-Wan Chung [2]

Div. of Computer Science, Dept. of EECS
Korea Advanced Institute of Science and Technology(KAIST)
373-1 Guseong-dong, Yuseong-gu, Daejeon, Republic of Korea, 305-701
[1]jpark@islab.kaist.ac.kr [2]chungcw@islab.kaist.ac.kr

***Abstract*— In this paper, we present iBroker, an Intelligent Broker for ontology based publish/subscribe systems which syntactically and semantically match incoming OWL data to multiple user profiles. iBroker effectively manages user profiles based on the semantics of query patterns in user profiles formulated in SPARQL. iBroker uses a semantic matching algorithm to efficiently process OWL data and generate the complete results for user profiles, considering the core semantics of OWL. Experimental results demonstrate that iBroker is more efficient and scalable compared to an existing broker for ontology based publish/subscribe systems.**

I. Introduction

With the increase in the amount of data on the Internet, the amount of information available to users is rapidly increased. Among large amounts of information, it is a time-consuming task for users to find appropriate information. Instead, users may want to be informed of up-to-date and accurate information which is relevant to their interests. For example, a newspaper site publishes thousands of newspaper articles per day. If a user wants to find certain articles, the user should visit the site and find out the ones by searching the list of titles of newspaper articles. Instead, a newspaper alerting service can provide the list of articles to the user based on the user's subscribed interest whenever articles are published in the site.

In order to support such services, publish/subscribe systems have been developed. In a publish/subscribe system, subscribers (i.e., users) subscribe their interests to brokers as profiles, publishers provide their newly generated information to brokers in the form of events, and brokers notify the events to their interested subscribers. Recently proposed publish/subscribe systems are content based. In a content based publish/subscribe system, subscribers express their interests based on the content (i.e., the structure and data values) of events, and events containing the subscribed contents are delivered to the corresponding subscribers.

As XML becomes the standard data format for representing and exchanging data on the Internet, various researches on XML based publish/subscribe systems have been conducted. In such systems, XML documents are events and XPath or XQuery, well-known query languages for XML, is used to express subscribers' interests. Those researches mainly focused on how to efficiently process XML data against a set of XPath (or XQuery) profiles, only considering the syntactical (i.e., structure) information of XML data and user profiles. Without considering the semantics, the path expression '/A/B/automobile' in an XML document cannot be matched with the path expression '/A/B/car' in a profile.

To overcome this problem, several researches on ontology based publish/subscribe systems have been conducted [1], [2], [3]. In those researches, RDF [4] is used as publications and some matching techniques to efficiently process RDF graphs against user profiles are presented. Processing RDF graphs, instead of RDF data itself, might be impractical when large sized RDF data are provided from publishers since the complete RDF data must be provided to the broker and translated into the RDF graph for the matching process. Thus, streaming based processing of RDF data itself is more appropriate in the real time environment. Moreover, the semantics supported in those researches are insufficient.

Recently, OWL(Web Ontology Language) [5] was developed as a vocabulary extension to RDF and RDFS to increase the expressive power of ontology data. With such features, OWL is a recommended semantic markup language for publishing and sharing ontologies on the World Wide Web, by W3C(World Wide Web Consortium). Moreover, OWL was chosen as an ontology description language and SPARQL [6] as a query language for ontology data by W3C Semantic Web Activity. As a result, to provide higher selectivity of ontology data for the dissemination, a research on OWL based publish/subscribe systems which use SPARQL as their profiles and process OWL data in the streaming fashion is required.

In this paper, we present iBroker, an Intelligent Broker for OWL based publish/subscribe systems which syntactically and semantically match incoming OWL data to multiple user profiles. iBroker effectively manages user profiles based on the semantics of query patterns in user profiles formulated in SPARQL. iBroker uses a streaming based semantic matching algorithm to efficiently process OWL data and generate the complete results for user profiles, considering the core semantics of OWL, which are Class, subClassOf, Property, subPropertyOf, inverseOf, symmetric, and transitive. In our experiment, iBroker demonstrates significantly improved matching performance compared to an existing ontology based publish/subscribe system which does not support subPropertyOf and the semantics for individuals either. On the average, the

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.214

IEEE computer society

overall processing time of iBroker is about 15.4 times better than that of the existing system.

II. Related Work

S-ToPSS [1] extends the attribute based publish/subscribe system with capabilities to process syntactically different, but semantically-equivalent information by using the ontology.

In contrast to S-ToPSS, where the ontology is only used for checking synonyms and taxonomy, OPS [3] uses RDF graphs as its publications and subscriptions. OPS uses a subgraph isomorphism algorithm to match an RDF graph with RDF based profile graphs. Basically, OPS creates matching trees for candidate profiles, and partial matching results found during the traversal of the RDF graph are maintained in these matching trees. When the traversal is completed, OPS performs the verification process of matching trees to check whether matching trees are potentially the results of profiles.

G-ToPSS [2] is another ontology based system which uses RDF for its publication and subscription models. G-ToPSS utilizes a hash table for managing user profiles and uses this hash table for matching RDF graphs. Thus, as demonstrated in [2], the performance of G-ToPSS is better than that of OPS. However, G-ToPSS only considers subClassOf semantics by using a class taxonomy. This class taxonomy contains the hierarchical structure of classes and their individuals. A problem with the class taxonomy is that it requires an additional overhead to maintain such taxonomy up-to-date.

III. iBroker

A publish/subscribe system processes a sequence of events (e.g., OWL documents) to generate profile results for subscribers. Such events should be processed in the streaming environment since a matching process can be performed before the parsing of a document is done.

In iBroker, an OWL parser which works like a SAX (Simple API for XML) parser is used. At the beginning of an OWL document, the OWL parser generates a start() event. For each class, the OWL parser generates a class(C, I) event, where C is a name of the class and I is an individual of the class C. When a property is encountered, the OWL parser generates a property(P, D, R) event, where P is a name of the property, D is a domain individual of the property P, and R is a range individual of the property P. At the end of an OWL document, the OWL parser generates an end() event.

A. Data Structure of iBroker

An OWL document can be represented as a directed labelled graph. Thus, in this paper, we begin with an assumption that every node (i.e., subject, object) in a graph has a unique name by their corresponding URIs, and no two edges (i.e., property) between any two nodes can have the same label either, similar to [2]. This assumption leads to the development of a hash table based data structure for the matching process since each of the triples in the WHERE clause of a SPARQL query can be separated. Thus, iBroker utilizes a two-level hash table, named profileHash, for managing user profiles.

At the first level of profileHash, all the triples for classes and properties expressed in all the user profiles are stored using the name of a class or a property as the hash key. At the second level of profileHash, a list of all the user profiles containing this triple is maintained with some information obtained from this triple in each user profile.

Example 1: Consider the following three SPARQL queries:
q1 = (?x type Article)
q2 = (?x type Columnist)(?x Writes ?y)
q3 = (?x type Article)(?x articleName 'ICDE2009 CFP')
The corresponding profileHash is depicted in Fig. 1.

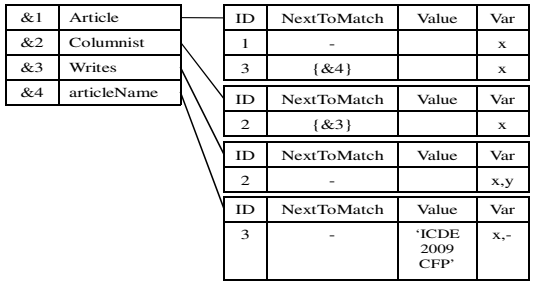

Fig. 1. An Example of profileHash for queries expressed in Example 1

Note that we only show the triples of the WHERE clause of a SPARQL query, for brevity. Each entry of the second level of profileHash contains the following four fields. The ID field indicates the number of a profile as a profile identifier. The NextToMatch field denotes that there is another triple to be considered for this profile to be matched. The Value field contains the value used in the triple of the SPARQL query. Lastly, the Var field contains variable names used in the triple.

```
Procedure BuildingProfileHash(P)
begin
1.  for each triple Ti in Profile P do {
2.      tripleInfo := profileHash.getTriple(Ti.name)
3.      if (tripleInfo is null)
4.          tripleInfo := profileHash.insertTriple(Ti.name)
5.      tripleInfo.insertTripleInfo(P.id, Ti.info)
6.  }
end
```

Fig. 2. The Algorithm for Building profileHash

Fig. 2 shows the algorithm for building profileHash. For each triple Ti in a profile P, a corresponding entry for Ti is checked from the first level of profileHash (Line 2). If there is no entry for triple Ti, the name of Ti is inserted into the first level of profileHash and the triple information of Ti is inserted into the second level of profileHash. When an entry for triple Ti exists in profileHash, the triple information of Ti is inserted into the second level of profileHash (Line (3)-(5)).

B. Semantic Matching Algorithm

The algorithm of the Semantic Matcher in Fig. 3 matches an incoming OWL document against user profiles. A boolean variable matchResult is used to indicate whether an event for a class is successfully matched to the entries in profileHash.

```
Procedure SemanticMatching(e)
begin
1.switch(e.type) {
2.  case start :
3.    create a match table for each class and property in profileHash
4.  case class :
5.    match := profileHash.getTriple(e.name)
6.    if (match is null)
7.      matchResult := false
8.    else {
9.      CName.Table + = (e.individual)
10.     matchResult := true
11.   }
12.   ancestorList := getClassAncestors(e.name)
13.   if (ancestorList is not null) {
14.     for each class Ci in ancestorList do {
15.       match := profileHash.getTriple(Ci)
16.       if (match is not null) {
17.         CName.Table + = (e.individual)
18.         matchResult := true
19.       }
20.     }
21.   }
22.case property :
23.   if (matchResult == true) {
24.     match := profileHash.getTriple(e.name)
25.     if (match is not null) {
26.       PName.Table + = (e.domain, e.range)
27.       type := obtainPropertyType();
28.       switch(type) {
29.         case inverse :
30.             inversePName.Table + = (e.range, e.domain)
31.         case symmetric :
32.             PName.Table + = (e.range, e.domain)
33.         case transitive :
34.             Compute the transitive closure in PName.Table
35.             Add new (domain, range) pairs in PName.Table
36.       }
37.     }
38.     ancestorList := getPropertyAncestors(e.name)
39.     if (ancestorList is not null) {
40.       for each property Pi in ancestorList do {
41.         match := profileHash.getTriple(Pi)
42.         if (match is not null)
43.             repeat Line (26)-(36)
44.       }
45.     }
46.   }
47.case end :
48.   GenerateResult()
49.}
end
```

Fig. 3. The Algorithm of the Semantic Matcher

For the start event, a table Table(individual) is created for each class and a table Table(domain, range) is created for each property in profileHash (Line (2)-(3)).

For each class event, a corresponding entry for class e.name is searched in the first level of profileHash. If there is no entry for e.name, the Semantic Matcher sets the boolean variable matchResult to false. Otherwise, the Semantic Matcher stores an individual e.individual into the corresponding result table and sets the boolean variable matchResult to true (Line (5)-(11)). For the subclass matching, the Semantic Matcher retrieves the corresponding ancestor list for class e.name. Then, the Semantic Matcher searches for an entry in profileHash and stores the individual e.individual into the result table for all matched ancestor classes in ancestorList (Line (12)-(21)).

For each property event, the value of the boolean variable matchResult is considered first (Line (23)). If it is false, the matching is not performed for property e.name. Otherwise, a corresponding entry for property e.name is searched in profileHash. If so, a (domain, range) pair of property e.name is stored in the corresponding match table (Line (24)-(26)). The Semantic Matcher obtains the type of this property (Line (27)) and performs the semantic matching according to the type of this property (Line (28)-(36)). The Semantic Matcher performs the subproperty matching in Line (38)-(45) for all ancestor properties of property e.name.

Lastly, when an event is END, the Semantic Matcher generates results by invoking GenerateResult() (Line (47)-(48)). Fig. 4 shows the algorithm for GenerateResult(). For each profile Qj of class Ci in profileHash, if an entry for the NextToMatch field is null, the match table becomes the result for profile Qj (Line (3)-(4)). Otherwise, the result for profile Qj is obtained by joining all the match tables of NextToMatch fields while navigating NextToMatch fields (Line (6)-(10)).

```
Procedure GenerateResult()
begin
1.for each class Ci in profileHash do {
2.   for each profile Qj in the entry for Ci do {
3.     if (Qj.NextToMatch is null)
4.       Qj.result := Ci.Table
5.     else {
6.       Qj.result := Ci.Table
7.       while (Qj.NextToMatch is not null) {
8.         Qj.result := Qj.result ⋈ Qj.NextToMatch.Table
9.         Qj.NextToMatch++
10.      }
11.    }
12. }
13.}
end
```

Fig. 4. The Algorithm for GenerateResult()

IV. EXPERIMENTS

We implemented iBroker using Java. We also implemented G-ToPSS based on the algorithms provided in [2]. Our experiments were performed on 2.66GHz Pentium 4 with 1.5GB of main memory, running Windows XP.

Data Set We conducted the experiments using Lehigh University Benchmark Data (LUBM) (available in http://swat.cse.lehigh.edu/projects/lubm/index.htm). LUBM is widely used well-known ontology benchmark data. We generated various sizes of OWL data: 500KB, 1MB, 5MB, 10MB, 50MB, and 100MB.

Query Set Synthetic SPARQL queries are generated by a simple generator, implemented for the experiments. By setting the maximum number of triples as 6, the probability of containing values as 10%, and the probability of containing types only as 10%, we generated various sets of 100, 500, 1000, 5000, and 10000 user profiles. For the experiments, profiles are loaded into the hash tables for iBroker and G-ToPSS, respectively.

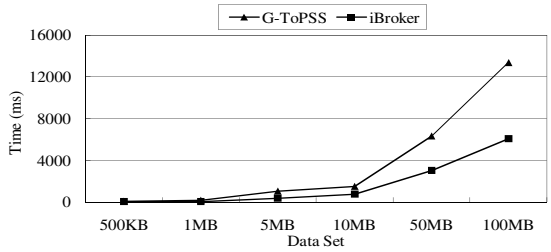

Fig. 5. Matching Time for 10000 User Profiles

Fig. 5 shows the matching times of G-ToPSS and iBroker for various sizes of incoming OWL data with 10000 user profiles. In G-ToPSS, there are more than one entries to be considered for matching a (domain individual, range individual) pair of a property since the hash table is constructed by using a (domain individual, range individual) pair of a property. This means that there are at least four entries to be considered for the matching. Moreover, G-ToPSS has to perform the matching for each property in incoming OWL data. In contrast to G-ToPSS, iBroker performs the matching for the names of classes and those of properties. However, there is only one entry to be considered for each class and property if there is no ancestors. Moreover, iBroker does not have to perform any matching for some properties when the class used as to define their domain is not matched. This feature of iBroker can enable the early pruning of unnecessary OWL data, similar to [7].

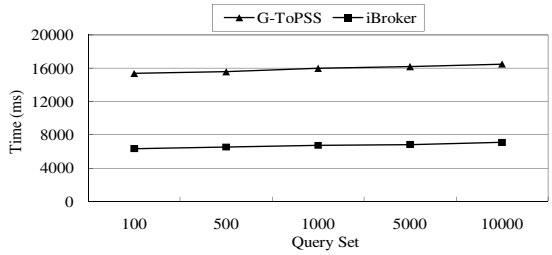

Fig. 6. Matching Time for 100MB OWL Data

Fig. 6 shows the matching times of G-ToPSS and iBroker for various sets of user profiles with the 100MB size of incoming OWL data. Similar to the above case, iBroker performs better than G-ToPSS.

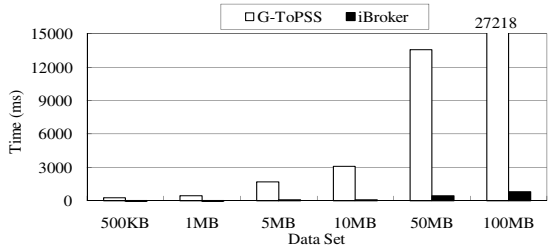

Fig. 7. Result Generation Time for 100 User Profiles

Fig. 7 shows the result generation times of G-ToPSS and iBroker for various sizes of incoming OWL data with 100 user profiles. When generating a result for a profile, G-ToPSS has to perform the join for the properties expressed in the profile and check the class of each joined individual in a class hierarchy by traversing the class taxonomy tree. Moreover, there would be numerous unnecessary individuals which are joined as the result before checking the hierarchical information of the domain class. In contrast to G-ToPSS, in iBroker, the match table for a property does not contain any (domain individual, range individual) pair of the property when the class used to define its domain is not matched. As a result, the result generation time of iBroker is significantly reduced compared to that of G-ToPSS.

Consequently, iBroker performs better than G-ToPSS for all the cases. On the average, the matching time of iBroker is 2.14 times faster than that of G-ToPSS and the result generation time of iBroker is 37.3 times faster than that of G-ToPSS. Lastly, the overall processing time of iBroker is about 15.4 times better than that of G-ToPSS.

V. Conclusion

In this paper, we proposed iBroker, an OWL based publish/subscribe system. iBroker manages user profiles formulated in SPARQL queries in the hash table, profileHash. iBroker also supports the semantic matching technique which increases the selectivity for the dissemination since inverse, symmetric, and transitive semantics can cause the new (domain, range) pairs to be matched. Thus, iBroker can match incoming OWL data against user profiles with higher selectivity. The experimental results show that the performance of iBroker is significantly improved compared to an existing ontology based publish/subscribe system.

Acknowledgment

This research was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement. (grant number IITA-2008-C1090-0801-0031)

References

[1] M. Petrovic, I. Burcea, and H.-A. Jacobsen, "S-topss: Semantic toronto publish/subscribe system," in *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2003, pp. 1101–1104.

[2] M. Petrovic, H. Liu, and H.-A. Jacobsen, "G-topss: Fast filtering of graph-based metadata," in *Proc. of the International World Wide Web Conference (WWW)*, May 2005, pp. 539–547.

[3] J. Wang, B. Jin, and J. Li, "An ontology-based publish/subscribe system," in *Proc. of the ACM/IFIP/USENIX International Conference on Middleware*, Oct. 2004, pp. 232–253.

[4] F. Manola, E. Miller, and B. McBride. (2004, Feb.) Rdf primer. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/rdf-primer

[5] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. (2004, Feb.) Owl web ontology language reference. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/owl-ref

[6] E. Prudhommeaux and A. Seaborne. (2008, Jan.) Sparql query language for rdf. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query

[7] M. M. Moro, P. Bakalov, and V. J. Tsotras, "Early profile pruning on xml-aware publish-subscribe systems," in *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2007, pp. 866–877.