DELIS
Dynamically Evolving, Large-scale Information Systems

Integrated Project
Member of the FET Proactive Initiative Complex Systems

Deliverable D2.3.5

Scalable P2P reasoning
Start date of the project: January 2004
Duration: 50 months
Project Coordinator: Prof. Dr. math. Friedhelm Meyer auf der Heide
Heinz Nixdorf Institute, University of Paderborn, Germany

Due date of deliverable: December 2007
Actual submission date: December 2007
Dissemination level: PU – public

Work Package 2.3: Global management of competing transport services over large net domains
Participants: RadioLabs (RAL), Universita di Roma “Tor Vergata”, Italy
Technical University Berlin (TUB), Germany

Authors of deliverable: Fabrizio Davide (fabrizio.davide@telecomitalia.it)
Giovanni Cortese (g.cortese@computer.org)
Dominic Battré (dominic.batre@tu-berlin.de)
Felix Heine (felix.heine@tu-berlin.de)
André Höing (andre.hoeing@tu-berlin.de)
Odej Kao (odej.kao@tu-berlin.de)
1 Introduction

This document describes the work performed in Workpackage 2.3 related to scalable semantic reasoning in P2P networks.

Future, large scale distributed systems will have an huge demand to store and query information in a scalable way. Large-scale network management systems are one example, where meta-data about the participating entities has to be stored, updated, and queried. The Web itself, and especially the envisioned Semantic Web [BLHL01] is another example. In both scenarios, the focus lies on machine processable data, not machine understandable.

The data has to be encoded in some format, that allows machines to further reason about it. The Semantic Web has produced good standards in this direction, ranging from the resource description framework (RDF) [MM04] as a flexible basis, over RDF Schema (RDFS) [BG04], up to the various flavors of the Web Ontology Language (OWL) [MvH04]. However, the main open question here is how to design a world-wide distributed system that is able to perform the reasoning and querying in a scalable way.

It is important to stress that the real value of the reasoning mechanisms is only exploited when they are applied to the set of knowledge as a whole. This means that it is not sufficient to just do the reasoning over individual sources of information separately. It is rather necessary to take every bit of information into account that is available in a system when answering a specific query. This demand leads to further scalability issues.

On the other side of the problem, the question is which kind of reasoning is supported, i.e. how expressive is the underlying logic. RDFS, the base standard for reasoning in the Semantic Web, is rather limited in its reasoning capabilities. It basically supports to describe a class hierarchy and a role hierarchy. After that, instances and role fillers are promoted along these hierarchies. This allows e.g. to state that subDepartmentOf is a sub-role of subOrganizationOf. When a role filler faculty4 subDepartmentOf universityBerlin is given, the reasoner can infer that faculty4 is also a sub organization of universityBerlin.

However, for many applications these reasoning capabilities are not sufficient. Therefore, the OWL sub-languages Lite, DL, and Full have been standardized. These allow an increasing wealth of axioms to be formulated, that are then used in the reasoning process. This includes the possibility to define inverse roles, transitive roles, and complex concept definitions. An example is the definition that a smartphone is a mobile phone that also includes an organizer. With RDFS, it is only possible to derive that a smartphone is also an organizer. However, if some device is both classified as an organizer and a mobile phone, OWL style reasoning is needed to conclude that this device is a smartphone.

Previous research on reasoning in the Semantic Web has concentrated mostly on stand-alone reasoners. The focus of this research was to increase the reasoning capabilities, however typically with a loss in performance and scalability. The description logic [BCM+03] community has produced a large variety of logics that are typically evaluated using a so-called tableaux algorithm. This algorithm basically tries to proof conclusions by contradiction. However, it is unclear how such an algorithm can be extended to handle distributed collections of data without centralizing them in one place.

On the other side of the spectrum, p2p based RDF stores have emerged. Our BabelPeers system [DELIS-TR-0599] was one of the first, and the first one that also supported RDFS reasoning. Using a DHT based p2p network provides a scalable basis to store and query large amounts of RDF data. However, these approaches are limited to simple reasoning tasks. Thus our goal in this context was to advance the reasoning capabilities of BabelPeers to support more features of the OWL language.

In the following section, we give an overview of the BabelPeers system that is the basis for our work. In section 3, we present both the RDFS reasoning algorithm and the newer approach we developed recently. Finally, we reference the main reports and papers that describe our contribution in detail.
2 BabelPeers overview

This section briefly introduces our BabelPeers system. The BabelPeers goal is to integrate knowledge originating from different sources, i.e., different nodes in the p2p network. Thus it is possible to get results from the union of all knowledge in the network, including knowledge derived via logical reasoning using pieces that originate from different peers. Even broadcasting a query to all nodes in the network and collecting the results could not deliver the same quality of results one gets if the knowledge is integrated. Thus we need to disseminate the knowledge to well-defined nodes in order to be able to find and access it efficiently during query processing.

We use the resource description framework for knowledge representation. RDF is a very flexible framework. Each piece of information is encoded as a triple consisting of subject, predicate, object. As URIs are used to identify resources, these triples can be linked together in a world-wide unique manner. Thus putting together large amounts of RDF triples stemming from different sources typically results in a large graph that describes the overall knowledge.

For storing and integrating the triples, we use a p2p network based on distributed hash tables (DHT). In a DHT, each data item is associated with an identifier from an identifier space, e.g., $0, \ldots, 2^{128} - 1$. Each node in the network is responsible for a certain range of this identifier space. Every item to be stored is then pre-distributed to the node responsible for the identifier of the item. For fault tolerance, items are additionally replicated over multiple nodes. In our case where data items are RDF triples, we disseminate each triple to three different nodes based on its components subject, predicate, and object. Thus we can later access the triples even if only one of the components is known.

Updates and deletions are handled via a soft-state process. Thus, every triple carries an expiration time and is automatically removed from the network when it does not get refreshed. This means that the dissemination process runs periodically to keep the knowledge in the network up-to-date.

3 Algorithms

We now present two different approaches to reasoning in distributed RDF datastores. In the first subsection, forward-chaining RDFS rules is explained. After that, we focus on reasoning by query rewriting.

3.1 Forward chaining RDFS

For the RDF Schema reasoning, we follow a forward chaining approach. This means that we generate and store instances of every new triple which follows from the RDF Schema rules, like the additional triples that are generated by propagating an instance along the class hierarchy. Our dissemination scheme has the advantage, that all triples which are needed to do this forward chaining will be located on the same node. Thus, after dissemination, we can run the reasoning process on each node locally, generating the new triples. However, these newly generated triples are then disseminated to the network to be accessible via the standard indices over subject, predicate, and object.

The whole process is visualized in figure 1. Each node has some triples in its “local triples” store. These triples are disseminated via the p2p network to the responsible nodes, which store them in their “received triples” store (step 1). Using only the triples in the received store, RDF Schema reasoning is performed to generate new triples locally (step 2). The new triples are stored in the “generated triples” store, and then disseminated again over the network (step 3). At the target nodes, they are again stored in the received store, where they might fire new RDF Schema rules. The whole process terminates as soon as no more rules can be fired.
3.2 Reasoning through query rewriting

The reasoning algorithm shown in the previous section delivers best performance during query execution, as every conclusion is already calculated a priori. However, it is limited with respect to reasoning capabilities. When increasing the expressive power of the underlying logic, the number of valid conclusions rises and makes forward-chaining increasingly inefficient during insert and update of data. Also in highly dynamic environments where data changes often, forward-chaining is prohibitively expensive. Thus we were searching for alternatives which execute the reasoning during query evaluation.

The basic idea is to use query rewriting. This means that a query is rewritten into a set of queries that can be evaluated without reasoning and that deliver the same result than the original query with reasoning. Thus every relevant axiom in the knowledge base is checked against the query and possibly leads to modifications in the query or even to a new query version.

For this, the whole knowledge base is split (as usual in description logics) in an ABox that contains assertions about concrete objects, and a TBox, that contains general axioms. The query execution is
now split in two phases: in the first phase, the query is rewritten using the axioms in the TBox. In
the second phase, the rewritten query is executed against the ABox. The first phase does not need
the ABox at all, while the second phase does not need the TBox.

A common assumption in this context is that the ABox is orders of magnitude larger than the
TBox. In our p2p scenario, we use this assumption and broadcast the TBox to every node. However,
as the ABox is very huge, we cannot broadcast it. We rather distribute it over the p2p network as
described above.

Now, any peer can answer queries. It first rewrites the query using the locally available TBox.
This is done entirely without network interaction. After rewriting the query, its execution is planned.
This includes to decide whether parts of the query are evaluated on other peers that have more parts
of the needed data locally available. Finally, the query is executed. This step includes fetching data
from various nodes in the network. The whole process is shown in figure 2.

The logic which is supported by our system is a description logic dialect below OWL-DL. This is
due to the fact that today no other algorithms than tableaux algorithms are known that support
OWL-DL completely. However, as tableaux algorithms are not applicable in our p2p scenario, we
have to restrict the logic. In detail, we support the following constructs in our logic:

- **Equality of complex concepts**: Here, complex expressions can be used to describe concepts.
  Examples include the above mentioned equality of smartphones with organizers that are also
  mobile phones. Or the definition of an student that is a human that takes some course.

- **Subsumption of complex concepts**: This is comparable to the concept equality, however
  here only the weaker subsumption relationship is defined. Continuing the above example, one
  might decide to define that someone who takes a course is a student, but that the contrary
does not necessarily hold, because there are students that do not take any courses.

- **Subsumption of simple roles**: For roles, no complex constructors are allowed. Thus the
  role subsumption allows to build a role hierarchy comparable to RDFS.

- **Inverse roles**: This allows to define some role to be the inverse of another role. Thus one
  might have that worksFor is the inverse role for hasEmployee. After defining this axiom, it is
  sufficient to state in the ABox only one of the two possible role fillers, the other will be inferred.
  This definition also influences the role hierarchy.

- **Transitive roles**: This can be used to declare that a role is transitive. In combination with
  role subsumption, this can be very powerful. Consider e.g. a role hasChild and another role
  hasDescendent. To get the intuitive fact that a descendent is somebody who is connect via
  one or more hasChild relations, we only need two axioms in the TBox. The first one states
  that hasDescendent subsumes hasChild, making every child also a descendent. The second one
  states that hasDescendent is transitive.

- **Restricting the range and domain of a role**: This is used e.g. to state that the domain of
  the role takesCouse is the concept student. However, this is not meant as an integrity constraint
  in the sense of relational databases. Rather, it is used to infer the type of an object that is
  used within the domain or range of the role.

We do not describe the rewriting mechanism here in full detail, it is documented in detail in
[DELIS-TR-0600]. We rather stick to a small example. Consider the following scenario. We have
the definitions of hasChild and hasDescendent given as above. Additionally, we have the fact that
hasAncestor is the inverse role for hasDescendent. Now consider the following query asking for people
that have ancestors that where authors:

\[ q_0(x) \leftarrow hasAncestor(x, y), Author(y) \]
In the first step, the algorithm detects that hasAncestor must be transitive, as its inverse role hasDescendent is transitive. This changes the query by adding a transitive hull operator, denoted with ∗:

\[ q_1(\?x) \leftarrow \text{hasAncestor}^\ast(\?x, \?y), \text{Author}(\?y) \]

Now the inverse role hasDescendent is added. As the transitive hull must be computed using both the tuples from hasAncestor and hasDescendent together, we have to swap the order of columns in hasDescendent and then build the union with hasAncestor. Swapping the columns of a role’s relation is denoted with −.

\[ q_2(\?x) \leftarrow (\text{hasAncestor} \cup \text{hasDescendent}^-)^\ast(\?x, \?y), \text{Author}(\?y) \]

Finally, we integrate the sub-role hasChild of hasDescendent. This must also be included in the transitive hull computation:

\[ q_3(\?x) \leftarrow (\text{hasAncestor} \cup \text{hasDescendent}^- \cup \text{hasChild}^-)^\ast(\?x, \?y), \text{Author}(\?y) \]

\( q_3 \) is the final version of the query that is now executed using the distributed ABox.

### 3.3 Execution of the rewritten queries in an P2P environment

Executing the rewritten query in a p2p environment is challenging. In the p2p environment, network bandwidth is the most expensive resource, more expensive than cpu time or other factors like local memory. Thus, our algorithms target towards minimizing network transmission. In this way, they are comparable with query planning in distributed relational databases. However, there are differences. First of all, our “database” consists of large amounts of relations that only have one (concepts) or two (roles) columns. Thus, queries have typically large numbers of joins.

Furthermore, queries are generated through an algorithm and thereby have certain typical characteristics. The most important is that all queries in a query set originated from a single identical query, which makes them structurally very similar. Last but not least, we have the transitive hull computation which is not part of normal relational databases.

The output of the rewriting algorithm is a set of queries that all have to be executed. The final result will be the union of all these queries. The main challenge in the first execution phase is to build a query plan for the whole set of queries that exploits the overlaps between the queries in an efficient way.

Before diving into the details, we describe how queries look like. The basic elements of queries are disjunctions of either roles or concepts, possibly including filters that restrict the values of columns. These disjunctions are used to build conjunctions, which make up individual queries in the query set. The appearing roles might be decorated using the − or ∗ operator mentioned in the example of the previous section.

As the domain and range axioms for roles essentially lead to typing statements for the individuals appearing in those roles, they are reflected in the rewritten queries through special concepts. Consider e.g. a role \( R \) that has domain \( C \). Thus a query \( C(\?x) \) asking for instances of \( C \) also has to include those individuals in its result set that appear in the domain of \( R \). As a result, the query is rewritten into \( (C \cup R[1])(\?x) \). Here \( R[1] \) denotes a special concept that consists of all individuals in the domain of \( R \).

The evaluation starts with a preprocessing phase. In this phase, every role and concept is checked whether it is empty. Although this implies a DHT lookup per concept, the effort for this phase can be kept small by caching the results. In this context, “empty” means that there are no ABox assertions regarding this role or concept. As any sub-concepts are already included in the queries, we can simply drop empty concepts/roles from the queries. In case a whole disjunction runs empty we can
immediately delete the corresponding query, as it cannot have any results. After the simplification, we can reduce the query set by re-joining similar queries.

With the remaining query set, a plan is build to execute these queries using a heuristic algorithm. In the first phase, each pair of queries is checked for common sub-queries. All common sub-queries are recorded. In the second phase, each query is tested against all common sub-queries, and it is counted how often each of the common sub-queries appears in the queries. If there is more than one common sub-query, the set of common sub-queries is optimized recursively using the same algorithm. This yields a DAG of common sub-queries for the original query-set.

After these preliminaries, the actual execution is done. Here, each query is evaluated sequentially by executing relational operators. These fetch the existing values for concepts and roles from the p2p network for each disjunction. The values are aggregated using a union operator to reflect the disjunctions. The conjunctions are executed as joins between the existing sub-results. The order of the disjunctions within a query is important for performance.

In each step, the query processor first tries to use already evaluated sub-queries from a cache that stores the previous sub-results. A sub-query is selected that has the largest amount of matching query disjunctions with the not-yet evaluated part of the current query. If no sub-query with cached values can be found, then the sub-query that has the largest overlap with the remaining query is selected and executed. Only in case that no useful sub-query is found, we select a “fresh” disjunction and evaluate it directly to proceed with the query. This disjunction is chosen based on the size of the base relations, that is again fetched from the network and cached.

As the same algorithm is used to evaluate and cache the sub-queries, also sub-sub-queries are cached and reused. This results in a dramatic reduction of query evaluation effort.

The evaluation of a disjunction takes place in two different ways. When no transitivity is involved, it is possible to fetch each participating role or concept directly from the peer that stores the data. In order to make the bandwidth consumption as small as possible, filtering is performed directly on the target peer. Here we have to consider different kinds of filtering. When one of the two columns of a role is specified as a constant, the peer directly filters all rows that do not match this constant in the respective column. However, more efficient filtering can be done based on already known values for variables.

For this, consider the following query: memberOf(?, departement0), telephone(?, ?). The peer that stores the memberOf role locally filters those rows that do not pertain to departement0. Thus it sends a list of every member of departement0 to the source peer that processes the query. The peer that holds the telephone role also sends the complete list of everybody’s telephone numbers to the source peer. Here, the join is processed.

This behavior can be improved using bloom filters [Blo70]. Firstly, after transferring the data from one relation to the source peer, this peer builds a bloom filter that captures the existing values for the common variable ?. This filter is then sent to the other peer that uses it to make the transmitted data set smaller and thus to lower bandwidth consumption. Here, the question is how to decide which side of the join is fetched first. Our heuristic chooses the smaller relation. This leads to the situation shown in figure 3.

An interesting improvement (which is, however, currently not implemented in BabelPeers) would be to build bloom filters directly on both sides and to exchange the filters before sending the data to the source peer. This could lower bandwidth consumption further.

When transitivity is included, the situation is different. Here, no filtering is allowed before all participating roles are brought together. Consider e.g. the disjunction (subDepOf, subOrgOf)*(?, ?). The goal is to find the transitive closure of the sub-organization relationship. As the role subDepOf that captures sub departments is a child role for subOrgOf, also these tuples have to be processes. However, when there are known restrictions to either variable, it is not possible to apply these filters before computing the closure.

As in our setting the data for subDepOf and subOrgOf is distributed over two different peers, we
have to first collect all data in one place. To minimize traffic, we do not collect the data on the source peer. We rather select the peer that already holds the largest part of the data. In our example, this might be the peer that holds the subOrgOf role. This peer becomes responsible to evaluate the whole disjunction. It fetches the data from the other involved relations, then computes the transitive closure, and finally applies any filters like available bloom filters. The result of the filtering process is then sent to the source peer for further processing. Figure 4 gives an overview.

During the evaluation of atomic roles and concepts, also the special concepts $R[1]$ and $R[2]$ are evaluated directly at the peer that stores the relation, leading to optimized bandwidth consumption. Also inverse roles $R^{-}$ can be part of rewritten queries. They are executed by swapping the subject and the object of role’s triples. This is also done directly on the peer that holds the role’s or concepts’s data.
4 Main contributions

The overall approach to p2p based reasoning using query rewriting is documented in [DELIS-TR-0601]. The main rewriting algorithm is detailed in [DELIS-TR-0600], which is a paper currently under review for the ESWC 2008 conference. Furthermore, we have written a book chapter [DELIS-TR-0599] that gives an up-to-date overview of our BabelPeers systems. The chapter will appear in 2008.

References


