First Experiences with Load Balancing and Caching for Semantic Web Applications

Technical Report

Alissa Kaplunova, Atila Kaya, Ralf Möller

17th July 2006

Abstract

In our case study we investigate a server for answering OWL-QL queries with distinguished variables only (henceforth called OWL-QL$^-$). This server acts as a proxy that delegates queries to back-end DL reasoners that manage the KB mentioned in the query. This report describes load balancing and caching strategies in order to exploit previous query results (possibly produced by different users of the local site) in the presence of incrementally answered OWL-QL queries. In addition, the effects of concurrent query executions on multiple (external) inference servers and corresponding transmissions of multiple result sets for queries are discussed.

STS Institute for Software, Technology & Systems

TUHH Hamburg University of Technology

1 Introduction

In our work we consider applications which generate queries w.r.t. many different knowledge bases. We presuppose that for a particular KB there exists many possible query servers. In order to successfully build applications that exploit these KB servers, an appropriate middleware is required. In particular, if there are many servers for a specific KB, the middleware is responsible for managing request dispatching and load balancing. Load balancing must be accompanied by middleware-side caching in order to reduce network latency.

In our view the KB servers we consider are managed by different organizations and, maybe in the near future, each transaction (or query) requires some "payment" in case of a commercial environment. Therefore, DL applications used in some company need some gateway inference server that provides local caching (in the intranet) to: (i) reduce external queries and (ii) avoid repetitive external server access operations in case multiple intranet applications pose the same queries.

In our case study we investigate a server for answering OWL-QL$^-$ queries[1]. This server (called RacerManager) acts as a proxy that delegates queries to back-end DL reasoners (RacerPro servers) that manage the KB mentioned in the query and load KBs on demand. Figure 1 shows this scenario. Compared to previous versions, the functionality of RacerManager has been substantially enhanced. We address the problems of load balancing and caching strategies in order to exploit previous query results (possibly produced by different users of the local site). Caching is investigated in the presence of incrementally answered OWL-QL$^-$ queries. In addition, the effects of concurrent query executions on multiple (external) inference servers and corresponding transmissions of multiple partial result sets for queries are studied.

2 OWL-QL$^-$ Server as a Middleware

Reasoning over ontologies with a large number of individuals in ABoxes is a big challenge for existing reasoners. To deal with this problem, RacerPro supports iterative query answering, where clients may request partial result sets in the form of tuples. For iterative query answering, RacerPro can be configured to compute the next tuples on demand (lazy mode). Moreover, it can be instructed to return cheap (easily inferable) tuples first.

[1] OWL-QL$^-$ stands for OWL-QL with distinguished variables only.

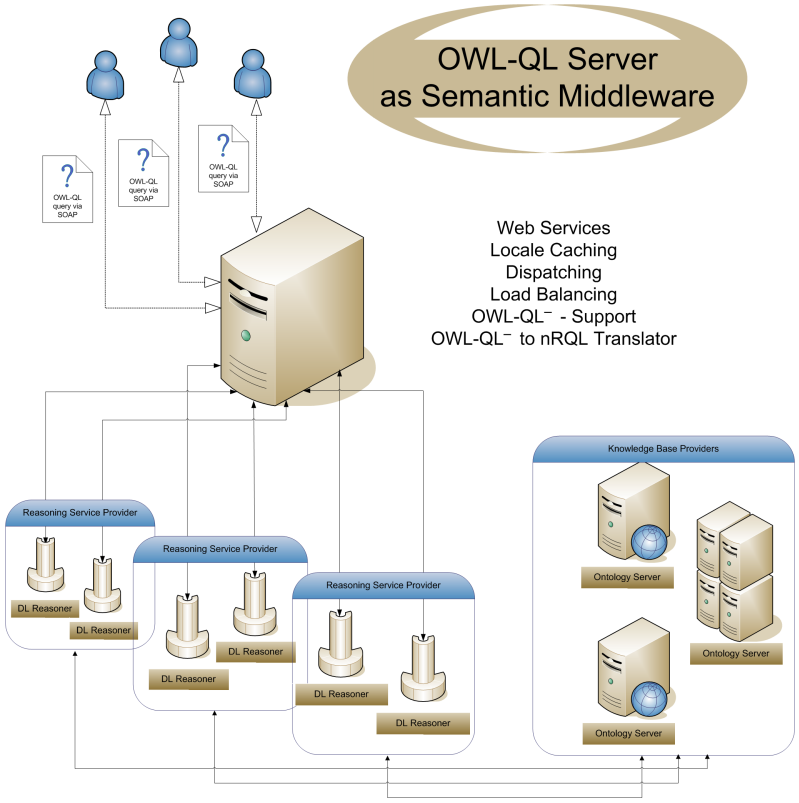

Figure 1: OWL-QL Server in the Semantic Web Scenario

Although these configuration options enable the reasoner to achieve significant performance improvements for a single client, this effect decreases in scenarios where multiple clients pose queries concurrently. In fact, a single RacerPro instance cannot process several client requests in parallel. Thus, as long as RacerPro is processing a clients request, which usually includes activities such as parsing the query, reading the corresponding knowledge base, classifying it, finding requested number of answer tuples and returning them, all other clients have to wait in a queue.

Motivated by the concurrency problem, our OWL-QL$^-$ server is implemented to act as a load-balancing middleware between clients and multiple RacerPro instances. RacerManager can initialize and manage an array of RacerPro instances. Multiple clients can use the web service offered by RacerManager to send their OWL-QL$^-$ queries concurrently. With respect to the states of the managed RacerPro instances and a naive load-balancing

strategy (similar to round-robin), RacerManager dispatches the queries to RacerPro instances. More precisely, given a query, which requires some ontology, RacerManager prefers RacerPro instances, which already worked on this ontology. Before a OWL-QL$^-$ query is send to a reasoner instance, it is translated to the new Racer Query Language (nRQL) by the translator module. Preliminary test results showed that, the proposed architecture prevents clients from blocking each other, as it is the case if multiple clients interact with a single reasoner.

Additionally, irrespective of load balancing and query dispatching, a client may benefit from the caching mechanism offered by RacerManager. In case he sends a query, which has been posed before, answer tuples are delivered directly from the cache. If the client requires more tuples than available in the cache, only the missing number of tuples are requested from an appropriate RacerPro instance. The cache expiration can be set to an arbitrary duration or turned off. In the latter case, the cache will never be cleared.

It is self-evident that a Semantic Web middleware has to support widely used and accepted standard components and protocols such as interfaces, query languages and communication protocols in order to support a wide range of applications. Consequently, we argue that service oriented architectures and web services are first class choices for a Semantic Web middleware. Moreover, OWL-QL is a candidate standard language and protocol for instance retrieval. Among other features it supports conjunctive queries. OWL-QL also defines a protocol for query-answering dialogues among agents using knowledge represented in the Web Ontology Language (OWL [10]). It facilitates the communication between querying agents (clients) and answering agents (servers). Due to the fact that huge amounts of information will be available in many different formats on the Semantic Web, OWL-QL offers features different from traditional database query languages such as SQL. In the Semantic Web scenario the amount of time a server needs to answer a query and the answer size may both become unpredictable. To overcome these problems, OWL-QL allows clients to specify the maximum number of answers they want to get from the server, and thus servers return partial sets of answers.

There are some recent prototypical OWL-QL server implementations, e.g., the Stanford OWL-QL Server [16] and DQL Server developed in Manchester [12]. The Stanford OWL-QL Server uses the first order logic theorem prover JTP [15] and the Inference Web [14] proof exchange system to answer the queries. It supports premises and proofs. The Manchester DQL Server implements the so called rolling-up technique to eliminate variables from a

query. For each new query, it requires all tuples from the reasoner at once and caches them.

Our system focuses on techniques which allow to achieve better scalability, high availability and the required quality of service. We forbear from implementing some OWL-QL features (e.g., may-bind variables) and rolling-up technique in the middleware tier, because we argue that only reasoners can fulfill these features efficiently.

The OWL-QL standard does not specify anything about server-side implementation details such as caching, number of reasoners used as back-end etc. However, the OWL-QL protocol defining query-answering dialogues and the heterogeneous nature of the Semantic Web makes it obvious that a middleware which claims to serve as an OWL-QL server has to manage multiple reasoners in the background. Using standard interfaces and a multi-layered architecture the middleware can be integrated with existing infrastructure such as firewalls, application servers or billing systems. This will enable OWL-QL servers to offer further services such as security, trust, accounting etc.

With respect to these challenges some of the crucial issues an OWL-QL server must support are: handling of concurrent client requests, management of multiple reasoners, request dispatching, load balancing and, finally, caching.

In the next sections we discuss these pivotal features of an OWL-QL server in detail.

2.1 Caching

Scalability is a crucial requirement for applications using database technology. Typically, database servers achieve this through replication of databases at the persistence layer, request dispatching and load balancing at the application layer or by caching at both layers.

In the Semantic Web scenario, reasoning is an expensive task that requires system resources and time. Nowadays most reasoners already implemented efficient caching mechanisms. However, if a new layer is involved in the scenario, namely a middleware that mediates between clients and reasoners, it is much more efficient to cache inferred knowledge in this layer. Caching in the middleware tier will avoid unnecessary communication with reasoners. This will get more important the more clients interact with the middleware. Furthermore, clients will benefit from knowledge gained through queries posed by other clients.

An OWL-QL server should cache each query sent by clients and each

Cache Algorithm: $getAnswerBundle(reqNumber) : tuples$
$reqNumber$ = number of tuples required by the client $lastNumber$ = number of the last tuple returned to the client $getCachedTuples(n)$ = procedure to get n tuples from cache $getNewTuples(n)$ = procedure to request n new tuples from reasoner $cacheSize$ = total number of tuples cached by the server for the given dialogue
1: **if** $(reqNumber + lastNumber) > cacheSize$ 2: **then** 3: $newTuples \leftarrow getNewTuples(reqNumber + lastNumber - cacheSize)$ 4: $cachedTuples \leftarrow getCachedTuples(cacheSize - lastNumber)$ 5: return $newTuples + cachedTuples$ 6: **else** 7: return $getCachedTuples(cachedSize - (reqNumber + lastNumber))$ 8: **endif**

Table 1: Cache Usage Algorithm

answer to that query gained through reasoning. Due to the nature of the OWL-QL specification, the middleware will probably not cache all results to a query but only some required subset of it. OWL-QL allows clients to specify the number of answers they want to get for their query as mentioned above. Whenever a client specifies the maximum number of results it wants to get, the OWL-QL server should return a process handle to the client. Afterwards the client can require more tuples to a query by using the received process handle. This means that the OWL-QL server must not only consider each query and its answers but also the identity of each client and the number of answers it already got.

The middleware should create a query-answering dialogue for each new query and cache answer tuples to this query. The next time, when the same query is posed by a client, the middleware will first look up in the cache of the corresponding dialogue to answer the query. Only if the required number of answers cannot be delivered completely from the cache, the corresponding reasoner will be contacted and only the missing tuples will be requested. This algorithm is presented in Table 1.

The described caching mechanism can reduce communication overhead with reasoners, particularly if they reside on remote servers, and therefore will improve overall system performance.

2.2 Request Dispatching and Load Balancing

On the one hand, in the context of business applications using database replication, queries are dispatched to a database instance with respect to the chosen load balancing strategy. On the other hand, instructions that require a change in data must be propagated to all database instances. To enable this, some databases, transaction processing monitors and application servers offer different mechanisms like distributed commit protocols.

In the case of Semantic Web applications using OWL-QL for querying, clients can only query knowledge bases but not modify them. Queries that include a premise, which are also called if-then queries, are no exception here, because they only require a temporary modification of the knowledge base. At first sight it looks like as dispatching and load balancing would be much more straight-forward for an OWL-QL server.

However, we have to consider the fact that clients can reference any KB in a query where they want to get results from. Therefore, an OWL-QL server has to track the state of each reasoner it manages. Using this information, the OWL-QL server can decide where to dispatch a query and can balance the load.

Whenever a query arrives at the OWL-QL server, the server has to check if it already processed this query and if some of the answers are already in the cache of the corresponding dialogue. Firstly, the server checks up its internal repository in order to find out if the knowledge base referenced in the query is already loaded by one of the connected reasoners. This means that the server already answered some queries from this knowledge base. In this case it has to inspect its state to see if a dialogue with the same query exists. If such a dialogue exists, the necessary answers will be taken from the cache. See Table 1 for details of cache usage. If none of the reasoners have the required KB loaded or there exists no dialogue with the same query, a new dialogue will be created. Details of the dialogue processing algorithm is shown in Table 2.

After a new dialogue is created or an existing dialogue is assigned, the server has to dispatch the query to the appropriate reasoner. The reasoner that already returned some answers to this query or at least loaded the referenced knowledge base is preferred. If no such reasoner can be found, the server has to delegate the required reasoning task to an idle reasoner with respect to some load balancing strategy (e.g., round robin). After an idle reasoner is found, the reasoner must be instructed to load the referenced knowledge base and then reason over it to return the required number of answers. See Table 3 for details of dispatching.

Dialog Algoritm: getDialog(query): dialog
1: **if** KBRegister.referencedKBLoadedOnAResoner(query) **then**
2: dialog ← searchDialogs(query)
3: **if** dialog == null **then**
4: return createNewDialog(query)
5: **else**
6: return dialog
7: **endif**
8: **else**
9: return createNewDialog(query)
10: **endif**

Table 2: Dialogue Processing Algorithm

Dispatch Algorithm: getReasoner(dialog): reasoner
1: **if** dialog.hasReasonerQueryId() **then**
2: return serverController.getReasoner(dialog.getReasonerQueryId())
3: **else**
4: return serverController.getNextIdleReasoner()
5: **endif**

Table 3: Dispatch Algorithm

3 Test Scenario

In order to validate our approach for an OWL-QL server discussed so far, we developed an open source system called RacerManager. Our application is implemented in Java and integrates several widely use components and systems, such as RacerPro as DL reasoner [6], Tomcat as application server/servlet container [13], Apache Axis Web Services Framework [2], XMLBeans Framework [3] and Jena Semantic Web Framework [9].

We chose a common n-tier architecture as the base layout for the system architecture. This is shown in Figure 2. This design provides a clear separation of responsibilities, easy extensibility by modification of single elements and a defined message flow through the system. The translator module converts OWL-QL queries to new Racer Query Language (nRQL) [7]. RacerManager can initialize and manage an array of RacerPro instances that are defined in a configuration file. Moreover, the cache expiration can be set to an arbitrary duration or turned off. In the last case, the cache will never be cleaned.

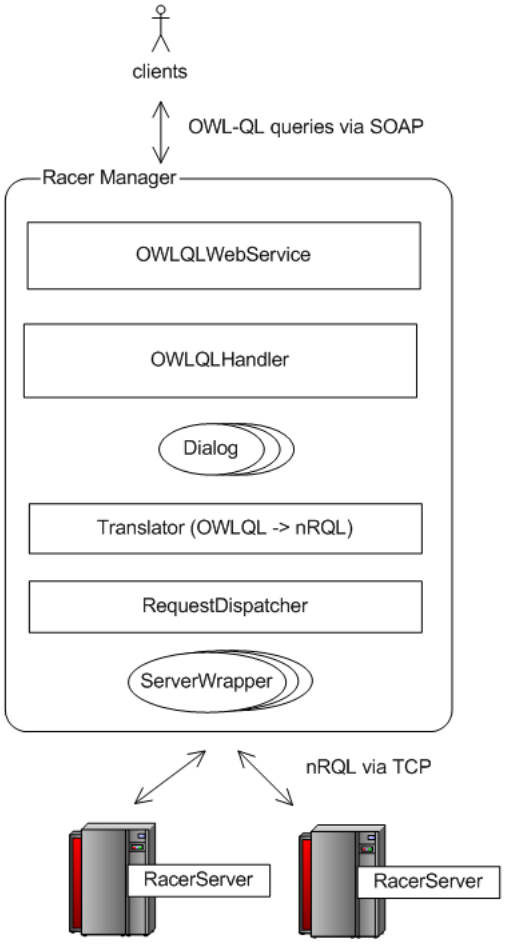

Figure 2: Architecture overview: RacerManager

Reasoning over complex ontologies or ontologies with a large number of individuals is a big challenge for existing reasoners. By using available configuration options, such as incomplete modes, that are powerful enough for semi-structured ontologies, RacerPro achieves significant performance improvements for handling large ABoxes. However, this task demands reasonable hardware resources and is especially memory-intensive.

In order to present a major benefit of our architecture with respect to large ABoxes and multiple clients querying the system concurrently, we tested RacerManager using the scenario shown in Figure 3.

In this scenario, a client sends Query 1 that requires reasoning over

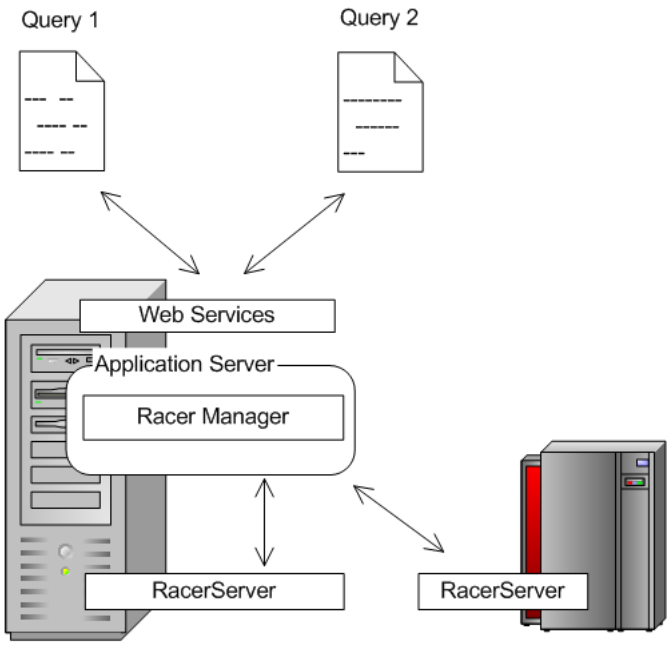

Figure 3: Test Scenario

an ontology with a large number of individuals. For our experiments we used an university ontology provided by the Lehigh University Benchmark. Query 1 results in more than 5000 individuals. RacerPro needs several seconds to answer this query. Concurrently, another client sends Query 2 which requires reasoning over a small sized ontology and RacerPro needs only some milliseconds to answer it. As shown in Figure 3 RacerManager is configured to manage one RacerPro instance on the local host and another one on a remote machine.

In this test scenario a client sends Query 1 to RacerManager first and few seconds later another client sends Query 2. As excepted, RacerManager

dispatches Query 1 and Query 2 to separate RacerPro instances that run on different machines. As a result the client sending Query 2 receives the desired results in a few seconds, whereas the first client with the more reasoning-intensive query receives its results later.

4 Features and Limitations

The target language of RacerManager is nRQL. OWL-QL and nRQL are both expressive languages that have a common subset but are difficult to compare. E.g., simple conjunctive ABox queries having only distinguished variables (must-binds) can be directly expressed in both languages. However, they both have some structures that are difficult to represent in the other one.

When developing the OWL-QL server, we not only considered the attributes of nRQL, but also the question if the middleware is the right place to implement all OWL-QL features. Some of them, such as may-bind variables or TBox related queries, could be implemented by the middleware but would result in excessive computing and increased communication with reasoners. Obviously, this would lead to suboptimal performance and poor scalability.

We presuppose that a superior number of Semantic Web applications will only require ABox reasoning through conjunctive queries with must-bind variables. Therefore, our current implementation supports such queries. Furthermore, it handles queries that include a premise (if-then queries).

The server does not support scenarios where multiple knowledge bases are to be used to answer a query or where a knowledge base is not specified by the client. Furthermore, RacerManager does not provide proofs about the reasoning made.

The following statements can be made about the conformance level of the server: The response collections returned by the server contain no duplicate answers. Therefore, the implementation can be called non-repeating. In the current version RacerManager automatically provides non-redundant answers because of the limitation to must-bind variables. However, this may change when the server is extended to support non-distinguished variables.

The OWL-QL specification defines a special termination token called *rejected* which indicates that the server is not able to answer the query for some reason. Our server returns a *rejected* token for features that are not yet supported (e.g., may-binds or dont-binds).

5 Conclusion and Discussion

In this paper, we proposed caching and query dispatching algorithms for a Semantic Web middleware. One of the instantiation of such middleware is RacerManager acting as an OWL-QL server. Results of our experiments show, that recent standard software engineering approaches, such as web services and service-oriented architecture, are also applicable in the Semantic Web domain.

It is obvious, that in some situations a single RacerManager server may itself become a bottleneck. One possible solution is to replicate it and to set up a HTTP or hardware dispatcher in front of RacerManager instances.

Considering the fact, that our main goal was to ensure scalability and high performance, we only taken into account the OWL-QL features, that could be efficiently implemented in the middleware tier. From our point of view, there are some open questions regarding the implementation of the OWL-QL standard which we will address in the future, e.g.: (i) How should queries without a KB reference be handled in different scenarios? (ii) In which scenarios should clients share results obtained by other clients? (iii) Is expiration of query results important for applications? (iv) Are subscription services required?

In our future work we will conduct comprehensive experiments in order to empirically evaluate RacerManager. Moreover, we will compare alternative strategies to determine more efficient load balancing and cache usage algorithms.

References

[1] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the semantic web. Technical Report KSL-03-14, Knowledge Systems Lab, Stanford University, CA, USA.

[2] Apache Foundation. Apache Axis Web Services Framework. URL, `http://ws.apache.org/axis/`.

[3] Apache Foundation. XMLBeans. URL, `http://xmlbeans.apache.org/`.

[4] DL Implementation Group. DIG interface API. URL, `http://dig.sourceforge.net/`.

[5] Yuanbo Guo, Jeff Heflin, and Zhengxiang Pan. Benchmarking daml+oil repositories. 2003.

[6] V. Haarslev and R. Möller. Description of the racer system and its applications. In *Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August*, pages 131–141, 2001.

[7] V. Haarslev, R. Möller, and M. Wessel. Querying the semantic web with racer + nRQL. In *Workshop on Applications of Description Logics, ADL '04*.

[8] Volker Haarslev, Ralf Möller, Ragnhild Van Der Straeten, and Michael Wessel. Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In *Proc. of the Int. Workshop on Description Logics, DL '04*.

[9] HP Labs. Jena Semantic Web Framework. URL, `http://jena.sourceforge.org/`.

[10] Deborah McGuinness and Frank van Harmelen. OWL web ontology language - overview. W3C Recommendation, URL `http://w3.org/TR/owl-features/REC-owl-features-20040210`, 2004.

[11] Miller, Swick, and Brickley. Resource description framework specification. URL, `htpp://www.w3.org/RDF/`.

[12] University of Manchester. Manchester DQL server. URL, `http://www.cs.man.ac.uk/~glimmbx/`.

[13] Apache Jakarta Project. Jakarta Tomcat. URL, http://jakarta.apache.org/tomcat/.

[14] Stanford University. Inference Web. URL, http://iw.stanford.edu/.

[15] Stanford University. JTP. URL, http://ksl.stanford.edu/software/JTP/.

[16] Stanford University. Stanford OWL-QL server. URL, http://onto.stanford.edu:8080/owql/FrontEnd.