Experiences with Load Balancing and Caching for Semantic Web Applications

Alissa Kaplunova, Atila Kaya, Ralf Möller
Hamburg University of Technology (TUHH)
al.kaplunova|at.kaya|r.f.moeller@tu-harburg.de

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. Therefore, DL applications used in some company need some gateway inference server that provides local caching (in the intranet) to: (i) reduce external communication 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. 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.

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.

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

Motivated by the concurrency problem, our OWL-QL$^-$ server is implemented to act as a load-balancing middleware between clients and multiple RacerPro instances. We chose a common n-tier architecture as the base layout. 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.

3 Features and Limitations

When developing the OWL-QL$^-$ server, our main goal was to ensure scalability and high performance. Therefore we only take into account OWL-QL features, that could be efficiently implemented in the middleware tier. In fact, some features such as may-bind variables implemented by the middleware using rolling-up techniques and disjunctive queries would result in excessive computing and increased communication with reasoners. In our opinion, they should be supported by reasoners directly.

We presuppose that for many Semantic Web applications, conjunctive queries with must-bind (distinguished) variables will be enough. 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.

4 Future Work

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.

It is obvious, that in some situations the 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.

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?