Enhancement of
the RacerProxy webservices Interface for
Iterative Query Answering

Tejas Doshi

submitted in partial fulfillment of the requirements for the degree
Masters of science in Information and Media Technologies

supervised by
Prof. Ralf Möller
Atila Kaya

Software Technology and Systems **(STS)**
Technical University of Hamburg-Harburg **(TUHH)**

Hamburg, January 2005

Thanks to

My Parents

For helping me start off with a good education,
from which all else springs...

Abstract

Racer-OWL-Webservice component is developed in the RACER 1.7.x Architecture to support OWLClients. There is an enhancement required in the Racer-OWL-Webservice to support new OWL query-answering dialogs, which is already supported by new RACER 1.8.x. This report describes how the enhancement being done.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 10th January 2005.

(Tejas Doshi)

Foreword

My formula to achieve success involves three D's - **D**etermination, **D**edication, and **D**iscipline. I have always believed in doing the most ordinary things in the most extraordinary way.

In an effort to continuous analyze strengths and weaknesses and redefine my capabilities, I realized the importance of maintaining equilibrium among all the ingredients to excel - intelligence, hard work, application of logic, systematic goal setting and exposure. Briefly, a desire to perform with extraordinary way enforced altogether different points of view inside me. I am always on the look out for improving myself. I strongly believed in exploiting my knowledge and my position to contribute for the betterment of humankind and improvement of the quality of the life on the planet.

"Every person have but one destiny and has to meet it". I have always been longing for expertise in a field that appeals me. Computers have always been fascinating thing for me and my fascination was boosted to a dream career due to my long time exposure to computers as my curriculum. I thoroughly witnessed the application of computers in process design, automation and auxiliary operations.

Symantic web technologies is the new edge in industry and research today. The more I have done projects, the more I have discovered my ignorance and my thirst for knowledge remains unquenched. I feel doing further projects on symantic web technology is the only way I can hone my skills, expand my knowledge and gain a new insight into the subject. This project is based on symantic web search and webservice technologies for a research project 'RACER' by Prof. Ralf Möller, Technical University of Hamburg-Harburg, Germany and Prof. Volker Haarslev, Concordia University, Montreal. Managing new, research based, and sophisticated technology requires a wide knowledge spanning across a few disciplines and a holistic approach towards the system. That's why I selected this project.

Tejas Doshi
tejas.doshi@tuhh.de

Content

At a Glance

Table of Contents

List of figures

Preface

Project Context

The World Wide Web (WWW) contains a large amount of information which is expanding at a rapid rate. Most of that information is currently being represented using the Hypertext Markup Language (HTML), which is designed to allow web developers to display information in a way that is accessible to humans for viewing via web browsers. While HTML allows us to visualize the information on the web, it doesn't provide much capability to describe the information in ways that facilitate the use of software programs to find or interpret it. The World Wide Web Consortium (W3C) has developed the Extensible Markup Language (XML) which allows information to be more accurately described using tags. As an example, the word Algol on a web site might represent a computer language, a star or an oceanographic research ship. The use of XML to provide metadata markup, such as Algol, makes the meaning of the work unambiguous. However, XML has a limited capability to describe the relationships (schemas or ontologies) with respect to objects. The use of ontologies provides a very powerful way to describe objects and their relationships to other objects in this new semantic web.

On 10th February 2004, the World Wide Web Consortium had announced final approval of two key Semantic Web technologies, the revised Resource Description Framework (RDF) and the Web Ontology Language (OWL). RDF and OWL are Semantic Web standards that provide a framework for asset management, enterprise integration and the sharing and reuse of data on the Web. These standard formats for data sharing span application, enterprise, and community boundaries - all of these different types of "user" can share the same information, even if they don't share the same software.

RACER is a server software for the semantic web developed by Prof. Ralf Möller (Technical University of Hamburg-Harburg, Germany) and Prof. Volker Haarslev (Concordia University, Canada). With RACER you can implement industrial strength projects and it is an ideal tool for research and development. RACER supports the W3C standards RDF/OWL.

The RACER Proxy allows you to access multiple Racer servers via a single proxy for load balancing and convenient access to the publish-subscribe interface for developing network-aware applications.

Jan Galinski is developing a webservice for facilitating web based OWLClient to use Racer (version 1.7.x), using own developed owl2nrql translator. Using this webservice OwlClient can query Owl to Racer server and can get all answers in Owl back at once.

Objective

OWL-QL is a formal language and precisely specifies the semantic relationships among a query, a query answer, and the knowledge base(s) used to produce the answer. Unlike standard database and Web query languages, OWL-QL supports query-answering dialogs in which the answering agent may use automated reasoning methods to derive answers to queries, as well as dialogs in which the knowledge to be used in answering a query may be in multiple knowledge bases on the Semantic Web, and/or where those knowledge bases are not specified by the querying agent. In this setting, the set of answers to a query may be of unpredictable size and may require an unpredictable amount of time to compute.

A query may have any number of answers, including none. In general, we cannot expect that a server will produce all the answers at once, or that the client is willing to wait for an exhaustive search to be completed by the server. We also cannot expect that all servers will guarantee to provide all answers to a query, or to not provide any redundant answers. OWL-QL attempts to provide a basic tool kit to enable clients and servers to interact under these conditions. Answers are delivered by the server in *bundles*, and the client can specify the maximum number of answers in each bundle also known as iterative query answering.

The specified Iterative query answering is now supported in new Racer (version 1.8.x). So now the objective is to enhance webservice also to support iterative query answering feature.

Major Benefits

- OWL-QL Query-answering dialogs will be supported for the web based OWLClients of Racer.
- Query answering performance will be enhanced because of the caching of answers in dialogs and dialogs can be shared by the client sessions having same query.
- Communication between webservice and Racer server will be decreased because of the caching.

Intended Audience

there are numerous papers describing how racer can be used to solve application problems. Without completeness one can summarize that applications come from the following areas:

- Semantic Web
- Electronic Business
- Medicine / Bioinformatics
- Natural Language Processing
- Knowledge-Based Vision
- Process Engineering
- Knowledge Engineering
- Software Engineering

Acknowledgements

I, hereby take an opportunity to express my gratitude towards courtesy extended to me by assigning the project, *Enhancement of the RacerProxy webservices Interface for Iterative Query Answering*.

Prof. Ralf Möller, Technical University of Hamburg-Harburg, for believing in me and keeping faith in my work, for giving me an opportunity to work in his research project.

Atila Kaya, for taking his precious time out of his busy schedule to discuss the real-time problems, and also for explaining me the insight of the existing and ongoing work of the project throughout this project development, for giving me a freedom to introduce my ideas and apply them, for guiding me throughout the development period towards the goal.

Jan Galinski, who gave me support to persevere through what seemed like a totally overwhelming and never-ending task, for giving me an inspiration and courage to work in the project, for discussing problems and exloring the way out. I couldn't have done it without him.

Jog Maharjan, for giving me support throughout the project in parallel design, development, and testing.

I also appreciate all the people of this project; I have left out the names of some of the people who helped me. For that I am sorry but nonetheless grateful for the many hearts and hands that made this project possible. Thank you for them, for their vision, their caring, their commitment and their actions.

And at last, I would like to thank my beloved problems which occurred on site for directing me to the goal of the complete solution.

Tejas Doshi
tejas.doshi@tuhh.de

Preliminaries 1

Basic terminologies which are essential to understand terms used in this report:

Semantic web

The Semantic Web is a mesh of information linked up in such a way as to be easily processable by machines, on a global scale. You can think of it as being an efficient way of representing data on the World Wide Web, or as a globally linked database.

The Semantic Web was thought up by Tim Berners-Lee, inventor of the WWW, URIs, HTTP, and HTML. There is a dedicated team of people at the World Wide Web consortium (W3C) working to improve, extend and standardize the system, and many languages, publications, tools and so on have already been developed. However, Semantic Web technologies are still very much in their infancies, and although the future of the project in general appears to be bright, there seems to be little consensus about the likely direction and characteristics of the early Semantic Web.

Ontology

An ontology is a specification of a conceptualization.

the term ontology means a specification of a conceptualization. That is, an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents.

Racer

For the processing of the knowledge available in the Semantic Web, inference engine is necessary. Inference engines deduce new knowledge from already specified knowledge.

Racer is a server software for the Semantic Web inference engine, also known as Racer server.

With Racer you can implement industrial strength projects and it is an ideal tool for research and development.

Racer Client

Any application which can send a request to Racer server and receive responses is Racer Client and from now I call it client only.

RacerProxy

The Racer Proxy, as shown in the figure below, allows you to access multiple inference engines via a single proxy for load balancing and convenient access to the publish-subscribe interface for developing network-aware applications.

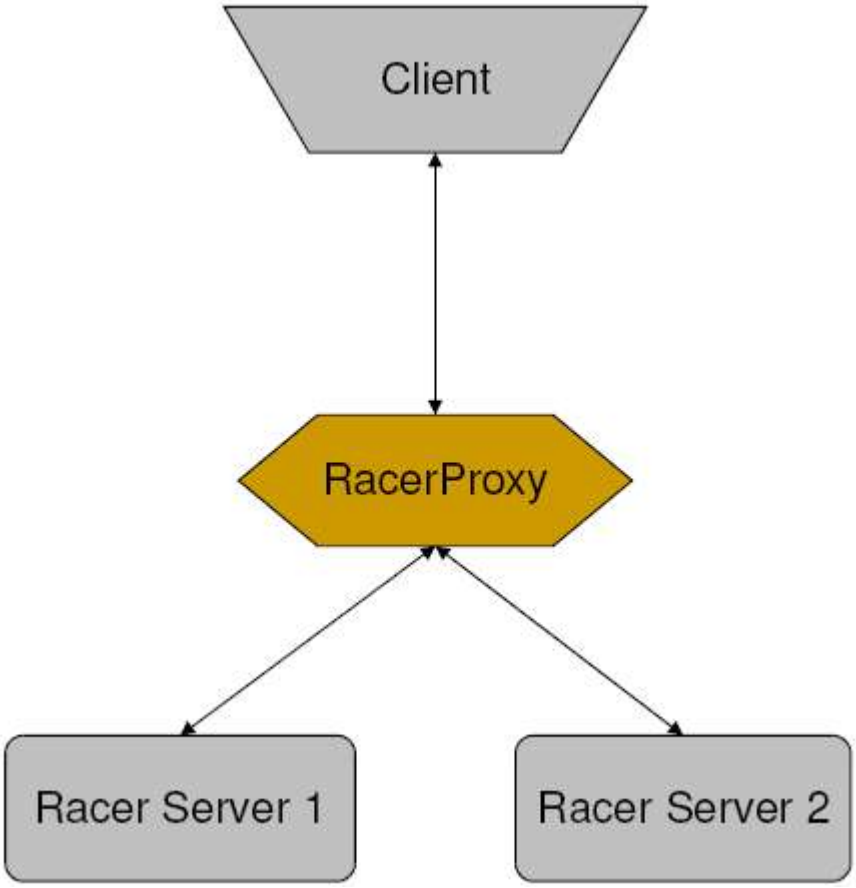


Figure 1.1 RacerProxy

OWL-QL

OWL Query Language (OWL-QL) is as a candidate standard language and protocol for query-answering dialogs among Semantic Web computational agents using knowledge represented in the W3C's Ontology Web Language (OWL). OWL-QL is a formal language and precisely specifies the semantic relationships among a query, a query answer, and the knowledge base(s) used to produce the answer. Unlike standard database and Web query languages, OWL-QL supports query-answering dialogs in which the answering agent may use automated reasoning methods to derive answers to queries, as well as dialogs in which the knowledge to be used in answering a query may be in multiple knowledge bases on the Semantic Web, and/or where those knowledge bases are not specified by the querying agent. In this setting, the set of answers to a query may be of unpredictable size and may require an unpredictable amount of time to compute.

Example:

```
<owl-ql:query
    xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
    xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:fam="file://famliy-1#">

<owl-ql:queryPattern>
    <rdf:RDF>
        <rdf:Description
            rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
            <rdf:type rdf:resource="file://family-1#woman" />
        </rdf:Description>
        <rdf:Description
            rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
            <fam:has-child rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#y" />
        </rdf:Description>
    </rdf:RDF>
</owl-ql:queryPattern>

<owl-ql:mustBindVars>
    <var:x/>
    <var:y/>
</owl-ql:mustBindVars>

<owl-ql:answerKBPattern>
    <owl-ql:kbRef
        rdf:resource="file://family-1#"/>
</owl-ql:answerKBPattern>

<owl-ql:answerSizeBound>2</owl-ql:answerSizeBound>

</owl-ql:query>
```

nRQL

An extended query language specially designed for Racer, called nRQL (for new Racer Query Language, pronounce: Nercle).

nRQL allows the use of variables and also allows to use complex queries.

Example:

- `(retrieve (?x) (?x woman))`
- `(retrieve (?mother ?child) (?mother ?child has-child))`

Webservices

Web Services are self-contained, modular applications that can be described, published, located, and invoked over a network, generally, the Web.

So using webservices one can publish the software service known by an URI, for use on the web. Any application can use xml based specification to know the interface of the service published and then can use the service on the web.

SOAP Message

A valid SOAP message is a well-formed XML document. SOAP is a specification for using XML documents as messages. The SOAP specification contains:

- A syntax for defining messages as XML documents, which we refer to as SOAP messages.
- A model for exchanging SOAP messages.
- A set of rules for representing data within SOAP messages, known as SOAP encoding.
- A guideline for transporting SOAP messages over HTTP.
- A conversion for performing remote procedure calls (RPC) using SOAP messages.

The SOAP specification defines a model for exchanging messages. It relies on three basic concepts: messages are XML documents, they travel from a sender to receiver, and receivers can be chained together. Working with just these three concepts, it is possible to build sophisticated systems that rely on SOAP.

The most fundamental concept of the SOAP model is the use of XML documents as messages. SOAP messages are XML. This provides several advantages over other messaging protocols. XML messages can be composed and read by a developer with a text editor, so it makes the debugging process much more simple than that of a complex binary protocol. As XML has achieved such widespread acceptance, there are tools to help us work with XML on most platforms.

We won't examine a SOAP message in detail but here is an example of one:

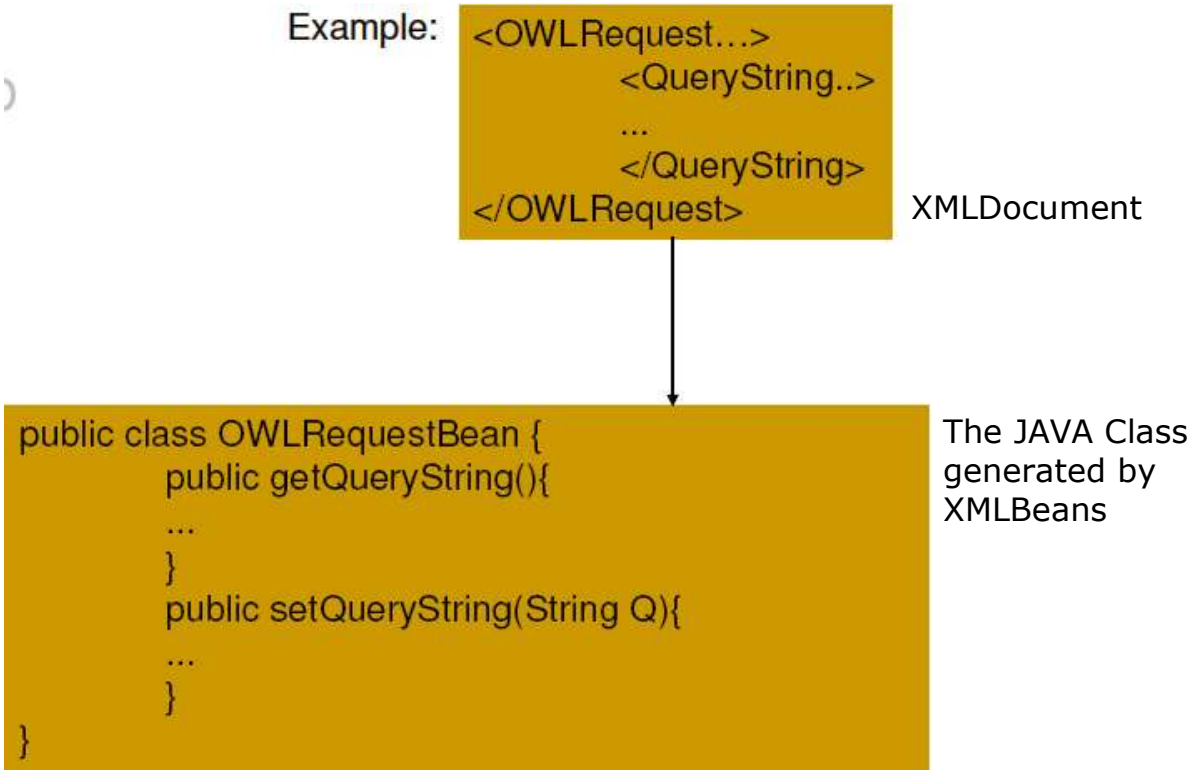
```
<soap:Envelope xmlns:soap”http://schemas.xmlsoap.org/soap/envelope/”
      soap:encodingStyle=”http://schemas.xmlsoap.org/soap/encoding/”>

   <soap:Header>
      <!-- SOAP Header is optional -->
   </soap:Header>

   <soap:Body>
      ...
   </soap:Body>

</soap:Envelope>
```

In short, we can compare SOAP message with a real world enveloped letter. The real world letter envelope includes all necessary information about the letter (message) like receiver's name, address, etc. However the message itself is in the letter (XMLDocument in the case of SOAPMessage).

XMLBeans

XMLBeans is a tool from apache that allows you to access the full power of XML in a Java friendly way. It is an XML-Java binding tool. The idea is that you can take advantage of the richness and features of XML and XML Schema and have these features mapped as naturally as possible to the equivalent Java language and typing

constructs. XMLBeans uses XML Schema to compile Java interfaces and classes that you can then use to access and modify XML instance data. Using XMLBeans is similar to using any other Java interface/class, you will see methods like *getFoo* or *setFoo* just as you would expect when working with Java. While a major use of XMLBeans is to access your XML instance data with strongly typed Java classes there are also API's that allow you access to the full XML infoset (XMLBeans keeps full XML Infoset fidelity) as well as to allow you to reflect into the XML schema itself through an XML Schema Object model.



It also works in the opposite direction, where you can use generated java classes to create an XMLDocument valid with respect to the given schema. Once the java objects are created , the tool serialize them into XMLDocument.

System Context 2

After an analysis of existing system I came to know the simple system context as figure 2.1:

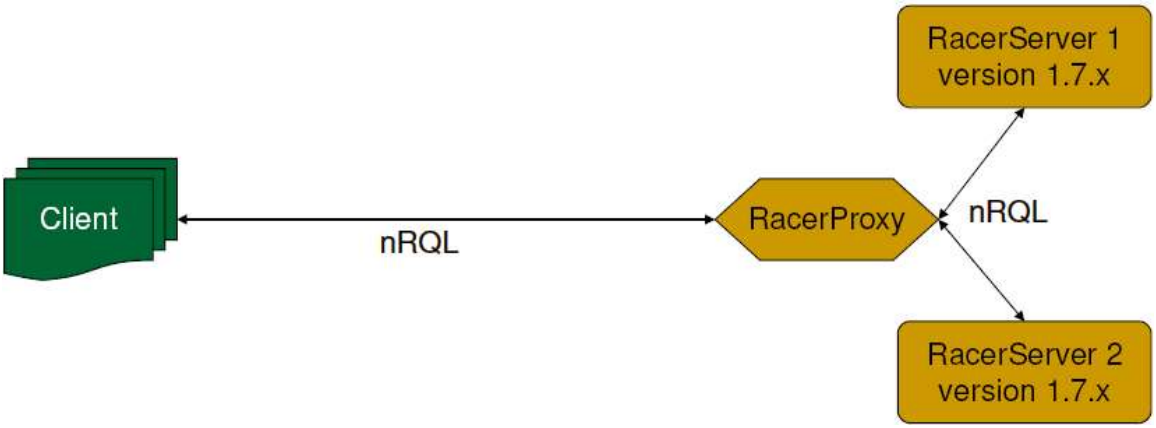


Figure 2.1 Simple system architecture

More than one Racer servers are connected to the racer proxy and communicate in nRQL. Client can connect to racer proxy and use racer for knowledge retrieval. Here we have clients that communicate in nRQL with racer proxy, so we call them nrqlClients.

As now W3C has declared OWL-QL as standard language to be used in sematic web applications. So we should also support owlClients in our system.

Jan Galinski developed owl-webservice for racer to support owlClients. He also developed owl2nrql translator to support this webservice. And so this looks like figure 2.2:

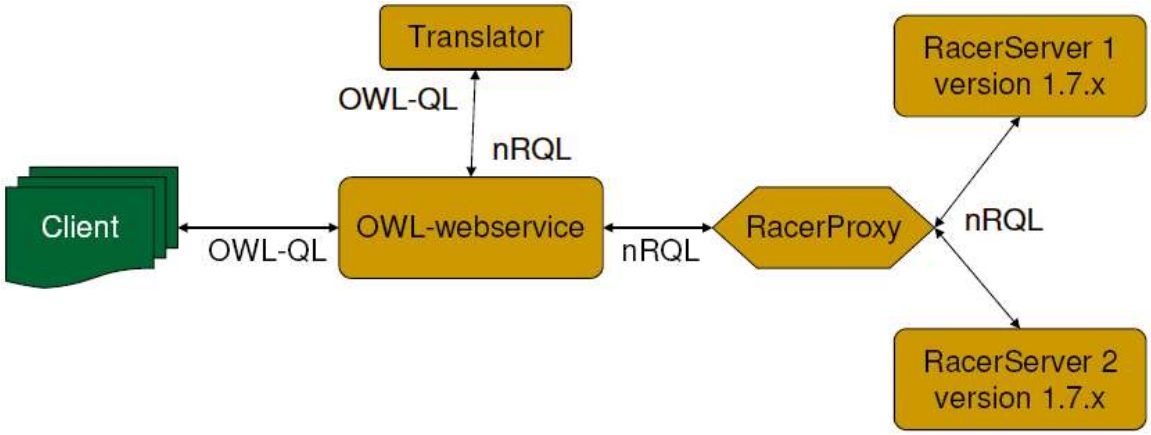


Figure 2.2 System architecture with owl-webservice

Now owlClients are supported and owlClient can ask for owlResponse after providing owlRequest to this owl-webservice. Owl-webservice then do translation from owlRequest to nrqlRequest and gets nrqlResponse and then after converting them to owlResponse can serve owlClient.

Here is an assumption that Racer server will respond all results in a set at a time and client will also wait for all results in a set at a time. This is not suitable for some situations like when racer server is busy and cant respond for sometime and when client is not interested in getting whole resultset but only first few chunks of results.

To solve this problem OWL-QL has developed a concept known as query-answering dialogs, described as following the details of OWL-QL.

OWL Query Language (OWL-QL) is a formal language and protocol for a querying agent and an answering agent to use in conducting a ***query-answering dialog*** using knowledge represented in the Ontology Web Language (OWL).

OWL-QL is a standard language and protocol for query-answering dialogs among Semantic Web computational agents during which answering agents (which we refer to as *servers*) may derive answers to questions posed by querying agents (which we refer to as *clients*). As such, it is designed to be suitable for a broad range of query-answering services and applications. Also, although OWL-QL is specified for use with OWL, it is designed to be prototypical and easily adaptable to other declarative formal logic representation languages.

A query may have any number of answers, including none. In general, we cannot expect that a server will produce all the answers at once, or that the client is willing to wait for an exhaustive search to be completed by the server. We also cannot expect that all servers will guarantee to provide all answers to a query, or to not provide any redundant answers. OWL-QL attempts to provide a basic tool kit to enable clients and servers to interact under these conditions.

Answers are delivered by the server in *bundles*, and the client can specify the maximum number of answers in each bundle. Each request from a client to a server for answers to a query can include an ***answer bundle size bound***, and the server is required to respond by delivering an ***answer bundle*** containing at most the number of query answers given by the answer bundle size bound. The collection of all answers sent to the client by the server in a query-answering dialog is called the *response collection* of that dialog.

An answer bundle must also contain either a ***process handle*** or one or more character strings called ***termination token****s*. The presence of a termination token in an answer bundle indicates that the server will not deliver any more answers to the query, and the presence of a server continuation in an answer bundle represents a commitment by the server to deliver another answer bundle if more answers to the query are requested by a client.

A client requests additional answers to a query by sending the server a ***server continuation*** containing the process handle provided by the server in the previously produced answer bundle and an answer bundle size bound for the next answer bundle to be produced by the server. Upon receiving a server continuation from a client, the server is expected to respond similarly by sending to that client another answer bundle. A client terminates a query-answering dialog by sending the server a

server termination containing the process handle provided by the server in the previously produced answer bundle.

Note that more than one client can participate in a given query-answering dialog with a server in that the client that sends a server continuation to the server need not be the same client that sent the original query or earlier server continuations during the dialog.

The overall structure of the dialog is illustrated in the following client-server communication steps:

- Client sends the request with the expected size of answer bundle.
- Server will respond the client with the given size of answer bundle, with the attached process handle.
- Now, if client has interest in getting more results from the server can use the process handle to send server continuation.
- Server will respond again with the given size of answer bundle.
- In our context we can see thes communications under one session. And there are two ways those can lead to end of this session.
- One, if server has no more answer to respond to client, server will return answer bundle with the termination token.
- Second, if client is not interested in getting more answers then it will send server termination to server to end the session.

These can be easily visualized in the following figure:

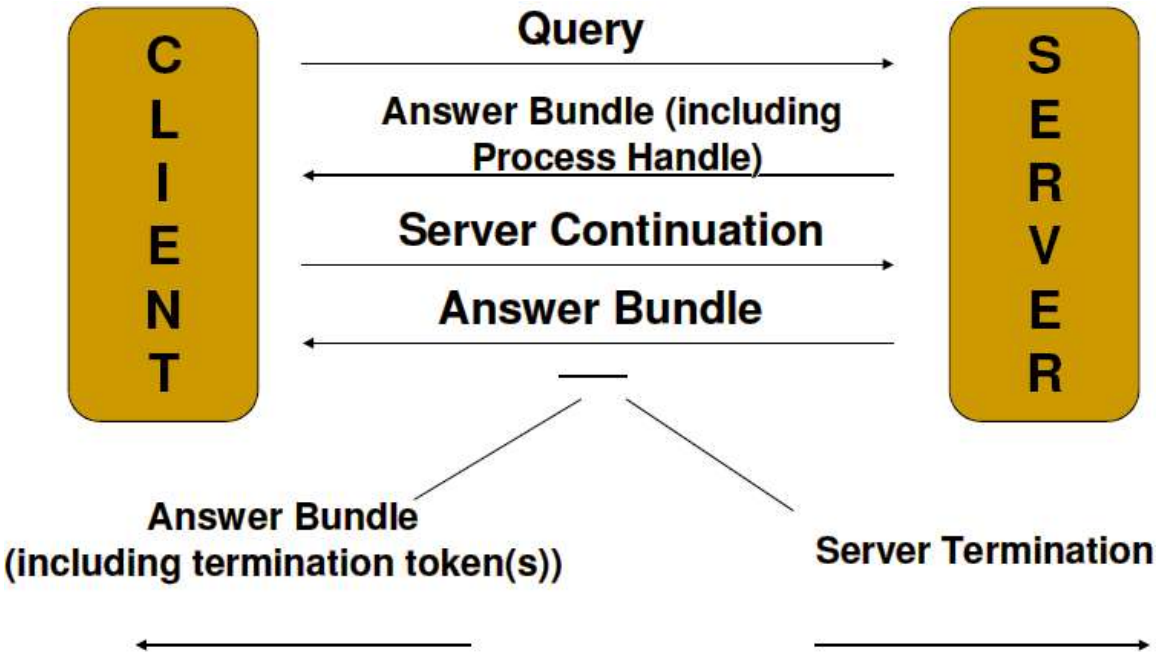


Figure 2.3 OWL-QL query-answering dialogs

The OWL-QL specification does not restrict the implementation of query dialog, the nature or content of process handles. They presented the bunch of ideas for solving problems and now its upto the project how it implements this concepts.

Different servers may use process handles in different ways. Some database servers may generate a complete table of answers, store it in association with a record of the query, and then use as a process handle an index or hash code keyed to the query record. Other servers may take advantage of the protocol to store enough information in a process handle to enable them to reconstruct the state of a search process and continue the search. Still others may simply store the answers already produced in a record of the query, use the query record as a process handle, and restart the query answering process from the beginning each time additional answers are requested. Note that the inclusion of a process handle in an answer bundle is not a commitment to provide more answers. If, for example, a server is unable to reconstruct the state of a query process when asked for more answers, it can always respond with an answer bundle containing a termination token and no answers.

Now the objective is to support this query-answering dialogs (also known as iterative query answering). For supporting this feature in our existing project, we need to enhance Racer server, Racer proxy and the owl-webservice.

Racer server has already been enhanced as in version 1.8.x, RacerProxy is in the enhancement phace now.

So with an assumption that owl-webservice has been developed and enhancement of racerProxy has been done. Now the enhancement of owl-webservice is required and that is the topic of this report. In the next chapter you come to know the solution of how this feature is adapted in the enhanced owl-webservice.

Final System Solution 3

Here, before describing the solution, it is important that it is assumed that Racer proxy is enhanced to support iterative query answering. Owl-webservice is developed to support owl-clients through the webservice interface.

Racer server 1.8.x is developed already to support iterative query answering. It is not released yet and available in a beta version.

Now I would like you to take to the final solution iteratively, in a way I got there.

Initial solution

First of all after initial analysis, I decided to have a chain of webservices. I thought to develop a webservice for query dialog management which will support owl-webservice in a chain to communicate with racerProxy.

In this solution, responsibility of the newly developed webservice are:
- To maintain query dialog information.
- To maintain session related information for each client.
- To maintain cache of results to uplift the performance benefits.

For each type of query communication we have *query dialog* and we share this query dialog between clients having same queries. So, we can use the cache of results to respond clients having same request. Note that, here client can participate in different query dialogs if he has multiple queries during one *session*. So, we have to store client session related information in query dialogs to know contextual information of the client session.

So, we will have multiple query dialogs, each of them is directly communicating with the proxy interface. Now, if there is no more client session to participate in particular query dialog then there is no need to keep such query dialogs in the memory and no need to keep the connection with the proxy alive. I thought of having a recycler thread which keeps running and watching on the query dialogs and if it founds any of them idle then it will kill that query dialog, I called it *dialog recycler*.

Then, instead of having query dialogs and dialog recycler as independent components, I thought of developing kind of a manager component which can manage all query dialogs and dialog recycler. I called it *dialog manager*.

All these components can be visualized easily altogether in the following figure:

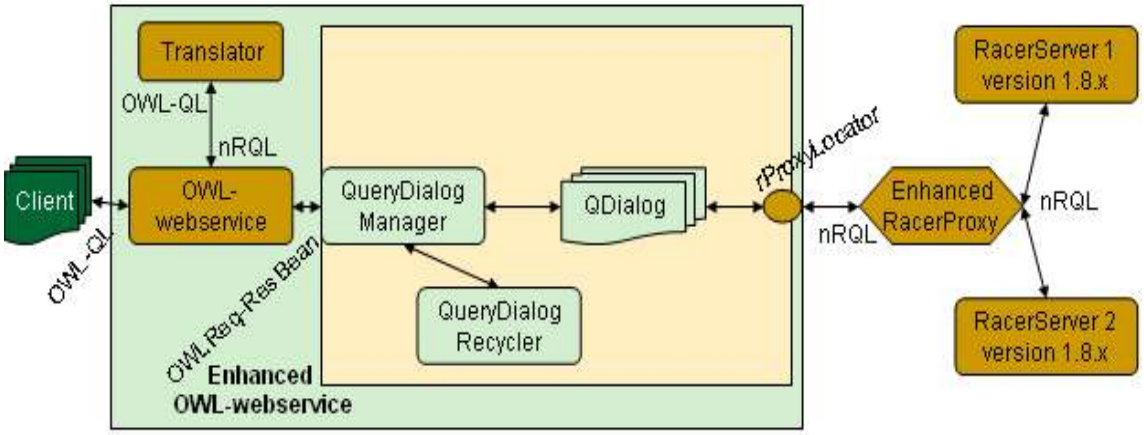


Figure 3.1 Initial solution

Now we can see that this can solve our problem. But, this solution is having some architectural weeknesses and I also solved them:

- Here there is no meaning of having chain of webservices, because this webservice will never going to be used directly by anyone.
- Always this service is going to used by the owl-service, there is no meaning of having this as a different service. So, I combined them in a one service.
- Here query dialogs have to keep all information about session related information and that is again a architectural defect. So, I kept all session related information in a session object.
- A session can participate in multiple query dialogs and many sessions can participate in one query dialog. So, we need to bind session with query dialog.
- We also need *session recycler* like dialog recycler, to keep watching all sessions and kill them if found idle.
- We also need some special manager to manage sessions and session recycler, I called it *session manager*.
- Then, we need a manager component which manages session manager and query dialog manager. I called it *racer-owl-manager.*

We can see how all these components can work together in the following figure as the final solution:

Final solution

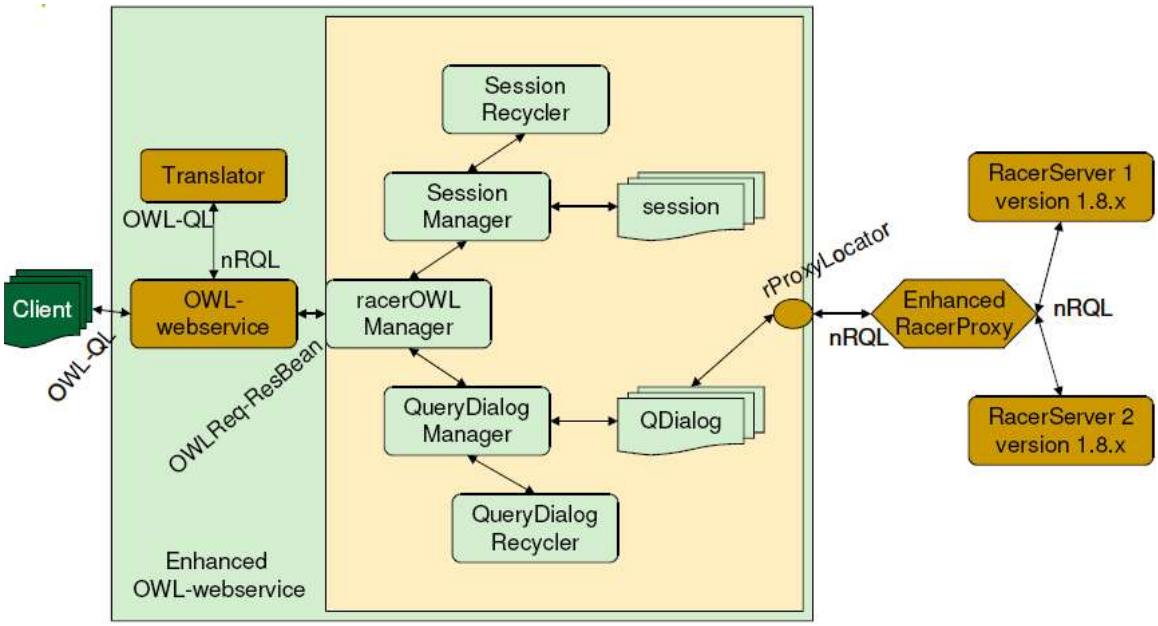


Figure 3.2 Final solution

RacerServer here used are version 1.8.x, which are enhanced now to support iterative query answering.

Enhanced RacerProxy is a newer version which is reengineered and enhanced from the previous version to support iterative query answering.

rProxyLocator is an interface provided by Enhanced RacerProxy to communicate with it.

Now we take a look at components, which together enhanced the owl-webservice.

QDialog is a query dialog from the above discussion and it is the component which keeps query dialog related information, makes connection with the Enhanced RacerProxy through rProxyLocator, and manages responses in a cache.

QueryDialog Recycler is a thread that keeps running and watching all QDialogs, if found any idle QDialog, then it will kill that QDialog and not let happen the waste of memory.

QueryDialog Manager manages all QDialogs and QueryDialog Recycler thread.

Session maintains all session related information for each client. Here session is not like an HTTPSession, but its a set of communication between client and server from start to end. That is why one client have only one session and this session can be binded to one or more QDialogs, and one or more sessions can also be binded to one QDialog to utilize the cache maintained by that QDialog.

Session Recycler is like an QueryDialog Recycler, is a thread that keeps running and watching all Sessions, if found any idle Session, then it will kill that Session and not let happen the waste of memory.
SessionManager is like QueryDialog Manager, manages all Sessions and Session Recycler thread.

RacerOWLManager is a main component in this design which manages SessionManager and QueryDialog Manager. SessionManager assisting it to create, to get, and also in to manage the sessions, as like QueryDialog Manager assisting it to create, to search, to get, and also in to manage the QueryDialogs. RacerOWLManager keeps the binding between sessions and QDialogs.

In this enhanced component, main responsibility of communicating with owl-webservice is of RacerOWLManager. Clients in this case send request as an OWL-QL enveloped in SOAP Message.

owl-webservice strips off that message and gets the OWL-QL request, converts that to nrql request, converts owl request to *OWLRequestBean,* attaches nrql request to that bean and pass that bean to RacerOWLManager.

RacerOWLManager then checks the session related information attached to that bean, if found not attached any information then tells SessionManager to create one for this client. SessionManager then creates and returns one session for this client and also checks that SessionRecycler is running or not. If not running then it starts that thread and it will keep running then after and keep watching all sessions. In the same case it then asks QueryDialog Manager to search matching QDialog for the current request. If found matching QDialog, it will return that to RacerOWLManager and if not then create one and return. QueryDialog Manager also checks the status of the QueryDialog Recycler and if found not running then starts that to keep watching on all QDialogs. Then RacerOWLManager bind the session with that QDialog and stores the binding information (this binding information will be returned with the response to the client and will be used by the clients in the continuation calls as a *process handle.* In the case of continuation RacerOWLManager will not create new objects but retrieve binded objects from process handle specified by the client).

In any case, QDialog makes a connection with RacerProxy using rProxyLocator interface. After getting connection it sends the request with the answer size bound. After getting answer bundle of size maximum which was sent with the request, stores those results in the cache managed by this QDialog. And then after only asks for more answer in a bundle if it cant respond from its cache.

In any case, Recycler threads are running and if found objects under watch idle for specified maximum time then they will kill that object and also notify about that to RacerOWLManager.

Client can have different requests in a single session, that is why one session can participate in many QDialogs, and many clients can have same request and that is why many session can share one QDialog and utilize the cache managed by the QDialog.

Following are the benefits of this solution, and in the next section we will see how the implementation has been done to achieve this solution design.

Software Implementation 4

In this section we will see how I implemented the final solution design which we have seen in the previous section. It was quite obvious how one can see the solution design. The following is the pseudo uml package diagram and class diagram showing implementation of the final solution design.

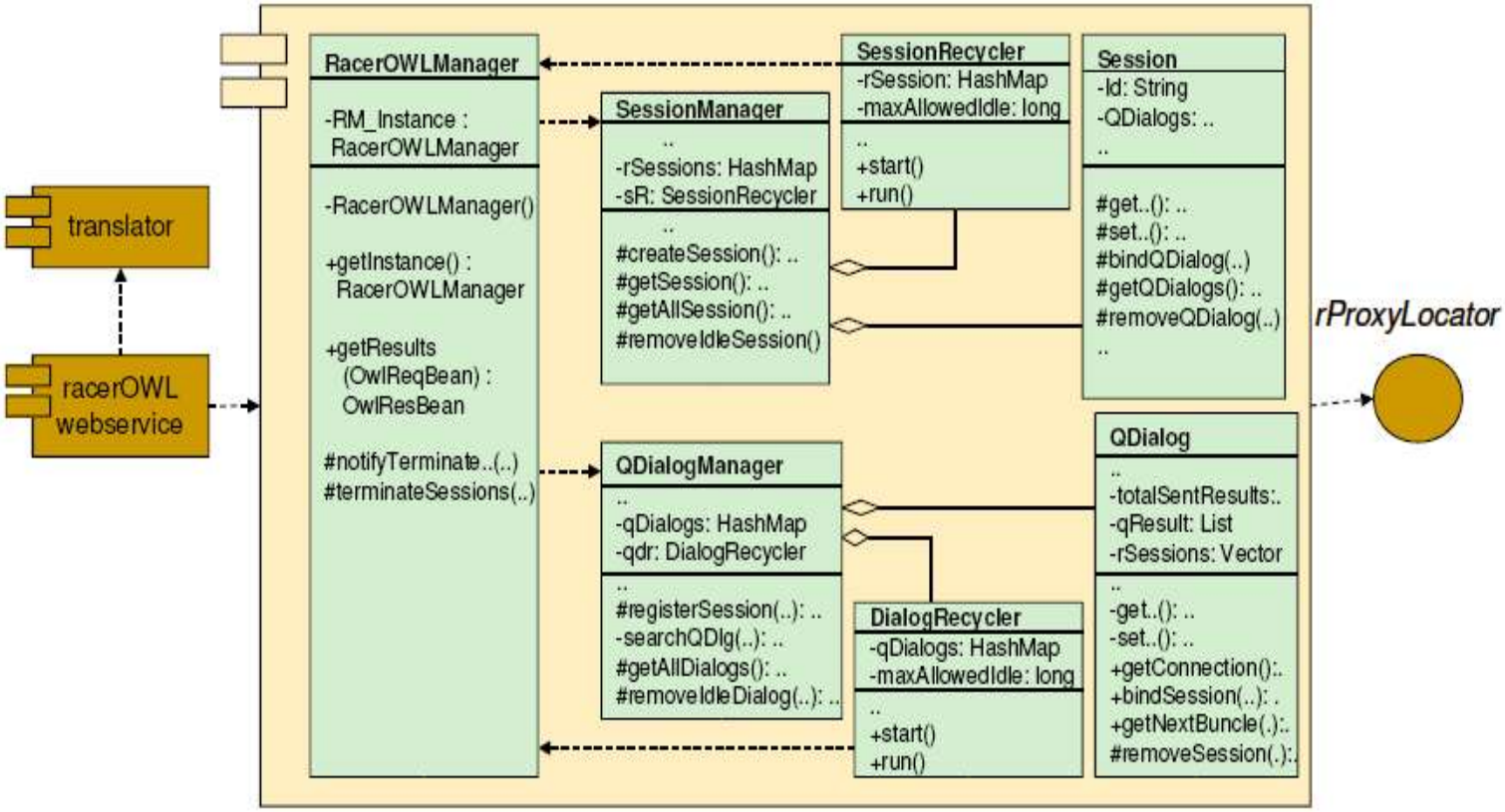


Figure 4.1 pseudo Package-Class Diagram - Final solution

Here it is assumed that owl-webservice is developed in a package, translator is also developed as an independent package, and racerProxy has provided the interface called rProxyLocator for communicating purpose with racerProxy.

Enhancement is done as an independent package, and I decided to make an own class for each component from the final solution design.

Relationships

Relationships between classes are important to know. And it will be clear after following statements describing relationships in words.

SessionManager contains all Sessions and SessionRecycler.
QueryDialog contains QueryDialogs and Dialog Recycler.

RacerOWLManager delegates session and query dialog related tasks to SessionManager and QueryDialog Manager.

SessionRecycler notifies RacerOWLManager about the killing of idle sessions.

Dialog Recycler notifies RacerOWLManager about the killing of idle Qdialogs.

Responsibilities

Now we will see responsibilities of each class implemented.

Session

Session keeps the following **data**:

Id Identical session ID.

created Timestamp of the session creation.

lastused Timestamp of the session used last time.

QDialogs vector of all QueryDialogs binded with this session.

Session offers following **operations**:

getId, setId: returns and sets the sessionID.

getCreated: returns the session creation timestamp.

getLastUsed, setLastUsed: returns and sets the session last used timestamp.

bindQDialog: binds the given QDialog with this session and add that QDialog to the vector of QDialogs binded with this sesion.

removeQDialog: removes binding of given QDialog from this session and also removes from the list of binded QDialogs with this session.

getQDialogs: returns the vector of QDialogs binded with this session.

SessionRecycler

SessionRecycler keeps the following **data**:

rSessions: hashmap of sessions to keep watching them for their idle duration, this list is continuously updated by the SessionManager.

maxAllowedIdle: threashold, which is maximum allowed idle time duration for the session, which is predefined.

SessionRecycler offers following **operations**:

SessionRecycler is a thread which keeps running and so does not offer any operation. It just automatically order sessionManager to recyle sessions if they are idle longer than the threshhold value.

SessionManager

SessionManager keeps the following **data**:

SM_Instance: singleton instance of the SessionManager.

rSessions: hashmap of all sessions.

sR: SessionRecyler.

sRunning: status of the SessionRecyler.

SessionManager offers following **operations**:

getInstance: returns the singleton instance of the SessionManager.

createSession: returns a session after creating a new one.

getSession: returns a session of given sessionID.

getAllSessions: returns a hashmap of all sessions.

removeIdleSession: removes the session of given sessionID of an idle session.

QDialog

QDialog keeps the following **data**:

QDID Unique query dialog identifier.

OQReq OWL-QL request associated with this query dialog.

NRqlRequest nRQL request translated from the OWL-QL request

created timestamp of the QDialog creation.

lastused timestamp of the QDialog used last time.

rSessions vector of all sessions binded with this query dialog.

qResult cache of the returned responses.

totalSentResults offset of sent results for each session binded with this query dialog.

rPLocator rProxyLocator to communicate with RacerProxy.

QDialog offers following **operations**:

getQDID, setQDID: returns and sets the unique query dialog identifier.

getOwlQlRequest: returns an OWL-QL request associated with this query dialog.

getCreated: returns a timestamp of query dialog creation.

getLastUsed, setLastUsed: returns and sets a timestamp of query dialog last used.

getConnection: gets a connection with the RacerProxy using rProxyLocator and returns the status of the connection.

bindSession: binds the given session with this query dialog and adds that to the vector of all sessions binded with this query dialog.

removeSession: removes the binding for the given session with this query dialog and also removes that from the vector of all sessions binded with this query dialog.

getAllSessions: returns a vector of all sessions binded with this query dialog.

getNextBundle: returns a response bundle of given maximum size or smaller. Tries to respond using cache, if not possible asks for the required responses to the RacerProxy using rProxyLocator and also manages the cache.

notifyBeforeKill: notify RacerProxy about killing of this query dialog before killling.

DialogRecycler

DialogRecycler keeps the following **data**:

qDialogs: hashmap of query dialogs to keep watching them for their idle duration, this list is continuously updated from the QDialogManager.

maxAllowedIdle: threashold, which is maximum allowed idle time duration for the query dialog, which is predefined.

DialogRecycler offers following **operations**:

DialogRecycler is a thread which keeps running and so does not offer any operation. It just automatically order QDialogManager to recyle query dialogs if they are idle longer than the threshhold value.

QDialogManager

QDialogManager keeps the following **data**:

QDM_Instance: singleton instance of the QDialogManager.

QDialogs: hashmap of the all query dialogs.

qdr: DialogRecycler.

qdRunning: status of the DialogRecyler.
QDialogManager offers following **operations**:

getInstance: returns the singleton instance of the QDialogManager.

registerSession: registers the given session with the given query dialog.

searchQDlg: returns the matching query dialog. Matching takes place among the all existing query dialogs and based on OWL string request. This method is only used internally.

equal: returns that given OWL string requests are equal or not. This operation is a supportive operation for the searchQDlg and only used internally. Here the comparision is based on simple string comparision and needed to be enhanced in future.

getDialog: returns the query dialog from the given unique query dialog identifier.

getAllDialogs: returns the hashmap of the all query dialogs.

removeIdleDialog: removes the query dialog based on the given unique identifier of the idle query dialog.

RacerOWLManager

RacerOWLManager keeps the following **data**:

RM_Instance: singleton instance of the RacerOWLManager.

RacerOWLManager offers following **operations**:

getInstance: returns the singleton instance of the RacerOWLManager.

getResults: returns the OWLResponseBean generated from the owl response for the given OWLRequestBean. getInstance and this are the only functions used by the webservices.

notifyTerminateQDialog: notified of the killing (termination) of a query dialog by this operation. Generally this operation is used by the DialogRecycler.

notifyTerminateSession: notified of the killing (termination) of a session by this operation. Generally this operation is used by the SessionRecycler.

terminateSessions: terminate all sessions from the given vector of sessions.

Demo Execution 5

Demo Case

Following are the considerations we need to do before understanding the demo execution and output.

- RacerServer 1.8.x is running.
- The famous family knowledge base has been loaded into the RacerServer.
- Enhanced RacerProxy has been deployed.
- Enahnced webservice has been deployed.

Now, after considering above situation, see that there are two clients (client1 and client2) having same request with different answer bundle size. Now follow the following execution steps and output chunks into the following figure to understand it thoroughly.

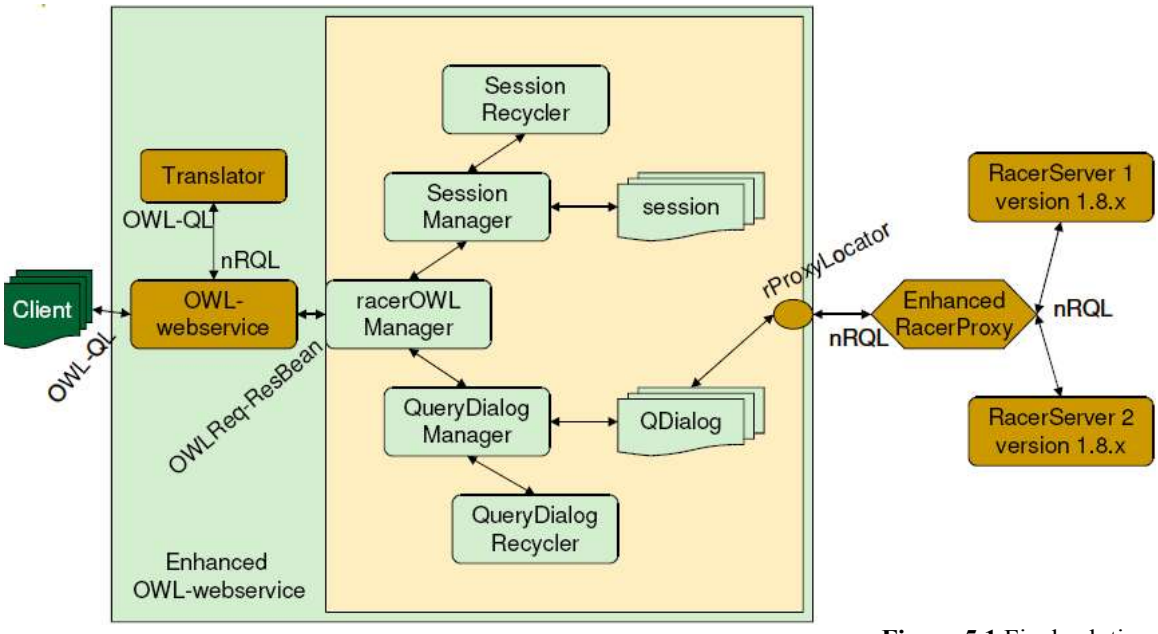


Figure 5.1 Final solution

· Client1 is sending a OWL request (retrieve women of the family, answer bundle size: 3) using the enhanced RacerOWLWebservice.

· RacerOWLWebservice is having OWLRequest enveloped in a SOAP message. It strips it and gets an OWLRequest, converts that to a nRQLRequest using

translator. Converts OWLRequest to OWLRequestBean, attaches a nRQLRequest to it and pass that on to the RacerOWLManager.

- RacerOwlManager checks the OWLRequestBean to see if there is any session information attached to it or not, if not like in this case, it will ask SessionManager to create one session for this client.
- SessionManager creates one session for this client and returns that to RacerOWLManager, it also starts the SessionRecycler thread because it is not running now. Thenafter SessionRecycler thread will keep on running and observing all sessions, if found any session is idle for a long time then will kill that session and notify RacerOWLManager about that.

Output:

```
##### CLIENT 1 #####
Session created..
```

- RacerOWLManager gives that session to bind with a matching query dialog to QDialogManager.

- QDialogManager will search through all of the query dialogs, and will not find any matching request in our case, create a new queryDialog (QDialog) and return that to RacerOWLManager for binding.

- RacerOwLManager does the binding, and store the binding information.

- QDialog makes a connection to the racerProxy using rProxyLocator and then sends the nRQLRequest with the answer size bound.

Output:

```
1/3/05 10:20:10 PM: Racer Proxy Started
1/3/05 10:20:10 PM: TCP-Connection to Racer localhost:8088 established
1/3/05 10:20:11 PM: EMail-Messanger started
1/3/05 10:20:11 PM: TCP-Messanger started
QDlg created..
```

- RacerProxy then connects to a Racer server and sends the query to get response from the RacerServer and after getting makes a bundle of nRQLResponse and returns it back to QDialog.

Output:

```
1/3/05 10:20:11 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 34
1/3/05 10:20:11 PM: TCP-Connector on port 7010 started
1/3/05 10:20:11 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:13 PM: ========================================
1/3/05 10:20:13 PM: [PROXY -> RACER] Set TUPLE-AT-A-TIME-MODE
1/3/05 10:20:13 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 19
1/3/05 10:20:13 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:15 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 27
1/3/05 10:20:15 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:17 PM: ========================================
1/3/05 10:20:17 PM: [WEBSERVICE -> PROXY] Connection received with query: (retrieve (?x) (?x woman))
1/3/05 10:20:17 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 28
1/3/05 10:20:17 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:19 PM: [PROXY -> WEBSERVICE] Connection successful

1/3/05 10:20:19 PM: [WEBSERVICE -> PROXY] Query received: [QUERY ID: 18306082,  maxBundleSize: 3 ]
1/3/05 10:20:19 PM: [PROXY] Query GET-NEXT-TUPLE: QUERY-2
1/3/05 10:20:19 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 27
1/3/05 10:20:19 PM: [RACER -> PROXY] message from racer localhost:8088 received
```

```
1/3/05 10:20:21 PM: [PROXY] Query GET-NEXT-N-REMAINING-TUPLES: QUERY-2 SIZE: 2
1/3/05 10:20:21 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 42
1/3/05 10:20:21 PM: [RACER -> PROXY] message from racer localhost:8088 received

1/3/05 10:20:23 PM: Proxy Data
Query ID:                QUERY-2
Query Dailog ID:         18306082
Racer ID:                0
MaxBundle:               3

1/3/05 10:20:23 PM: [PROXY -> WEBSERVICE] Response sent: QID QUERY-2
```

- QDialog caches those results and returns to RacerOWLManager, RacerOWLManager makes a OWLResponseBean out of that, attaches a processHandle to it and returns that to owl-webservice.

- owl-webservice translates that to OWLResponse and attaches header information, enveloped a SOAP message and returns it to Client1.

- Client1 got the results and his session has recorded the information about this communication.

Output:

```
##### CLIENT 1 - output #####
client1 Request1: (retrieve (?x) (?x woman)) answerBundleSize: 3
Response1:
((?X BETTY))
((?X ALICE))
((?X EVE))
##########
```

- Now Client2 is sending an OWL request (retrieve women of the family, answer bundle size: 2) using the enhanced RacerOWLWebservice.

- RacerOWLWebservice is having OWLRequest enveloped in a SOAP message. It strips it and gets an OWLRequest, converts that to a nRQLRequest using translator. Converts OWLRequest to OWLRequestBean, attaches a nRQLRequest to it and pass that on to the RacerOWLManager.

- RacerOwlManager see into the OWLRequestBean that is there any session information attached to it or not, if not like in this case, it will ask SessionManager to create one session for this client.

- SessionManager creates one session for this client and return that to RacerOWLManager.

Output:

```
##### CLIENT 2 #####
Session created..
```

- RacerOWLManager gives that session to bind with a matching query dialog to QDialogManager.

- QDialogManager search through all of the query dialogs, and found one in this case, return that queryDialog (QDialog) to RacerOWLManager for binding.

- RacerOwLManager do the binding, and store the binding information.

- QDialog look into the cache that it can respond back from the cache or not, it can in this case, retrieve response from the cache and return the bundle of the answer size bound to the RacerOWLManager.

- RacerOWLManager makes a OWLResponseBean out of that, attaches a processHandle to it and returns that to owl-webservice.

- owl-webservice translates that to OWLResponse and attaches header information, enveloped a SOAP message and returns it to Client2.

- Client2 got the results and his session has recorded the information about this communication.

Output:

```
##### CLIENT 2 - output #####
client2 Request1: (retrieve (?x) (?x woman)) answerBundleSize: 2
Response1:
((?X BETTY))
((?X ALICE))
##########
```

- Client2 has interest to get more results on this same query, so client2 send the OWL request to RacerOWLWebservice to get the next bundle with answer size bound and the processHandle from the previous response.

- RacerOWLWebservice is having OWLRequest enveloped in a SOAP message. It strips it and gets an OWLRequest, converts that to a nRQLRequest using translator. Converts OWLRequest to OWLRequestBean, attaches a nRQLRequest to it and passes that on to the RacerOWLManager.

- RacerOwlManager checks the OWLRequestBean to see if there is any session information attached to it or not, if it is attached like in this case, it will look for the processHandle and get to know about session and binded queryDialog out of that, and pass the request to that QDialog.

- QDialog checks the cache that it can respond back from the cache or not, it can not in this case, retrieves response from the cache and asks for the remaining answers (only one in this case) to the RacerProxy using rProxyLocator.

- RacerProxy then does the same to get response from the RacerServer and after getting makes a bundle of nRQLResponse and returns back to QDialog.

Output:

```
1/3/05 10:20:23 PM: [WEBSERVICE -> PROXY] Query received: [QUERY ID: 18306082,  maxBundleSize: 1 ]
1/3/05 10:20:23 PM: [PROXY] Query GET-NEXT-N-REMAINING-TUPLES: QUERY-2 SIZE: 1
1/3/05 10:20:23 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 42
1/3/05 10:20:23 PM: [RACER -> PROXY] message from racer localhost:8088 received

1/3/05 10:20:25 PM: Proxy Data
Query ID:                QUERY-2
Query Dailog ID:         18306082
Racer ID:                0
MaxBundle:               4

1/3/05 10:20:25 PM: [PROXY -> WEBSERVICE] Response sent: QID QUERY-2
```

- QDialog caches those results and makes a response bundle with no sent results and returns that to RacerOWLManager, RacerOWLManager makes a OWLResponseBean out of that, attaches a processHandle to it and returns that to owl-webservice.

- owl-webservice translates that to OWLResponse and attaches header information, enveloped a SOAP message and returns it to Client2.

- Client2 got the results and his session has recorded the information about this communication.

Output:

```
##### CLIENT 2 - output #####
client2 Request2: (retrieve (?x) (?x woman)) answerBundleSize: 2
Response2:
((?X EVE))
((?X DORIS))
##########
```

This is how the execution takes place, and try to utilize cache for better performance. You can see the whole output for the above demo execution on the following page.

Complete Output

```
##### CLIENT 1 #####
Session created..
1/3/05 10:20:10 PM: Racer Proxy Started
1/3/05 10:20:10 PM: TCP-Connection to Racer localhost:8088 established
1/3/05 10:20:11 PM: EMail-Messanger started
1/3/05 10:20:11 PM: TCP-Messanger started
QDlg created..
1/3/05 10:20:11 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 34
1/3/05 10:20:11 PM: TCP-Connector on port 7010 started
1/3/05 10:20:11 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:13 PM: ========================================
1/3/05 10:20:13 PM: [PROXY -> RACER] Set TUPLE-AT-A-TIME-MODE
1/3/05 10:20:13 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 19
1/3/05 10:20:13 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:15 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 27
1/3/05 10:20:15 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:17 PM: ========================================
1/3/05 10:20:17 PM: [WEBSERVICE -> PROXY] Connection received with query: (retrieve (?x) (?x woman))
1/3/05 10:20:17 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 28
1/3/05 10:20:17 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:19 PM: [PROXY -> WEBSERVICE] Connection successful

1/3/05 10:20:19 PM: [WEBSERVICE -> PROXY] Query received: [QUERY ID: 18306082,  maxBundleSize: 3 ]
1/3/05 10:20:19 PM: [PROXY] Query GET-NEXT-TUPLE: QUERY-2
1/3/05 10:20:19 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 27
1/3/05 10:20:19 PM: [RACER -> PROXY] message from racer localhost:8088 received
1/3/05 10:20:21 PM: [PROXY] Query GET-NEXT-N-REMAINING-TUPLES: QUERY-2 SIZE: 2
1/3/05 10:20:21 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 42
1/3/05 10:20:21 PM: [RACER -> PROXY] message from racer localhost:8088 received

1/3/05 10:20:23 PM: Proxy Data
Query ID:                QUERY-2
Query Dailog ID:         18306082
Racer ID:                0
MaxBundle:               3

1/3/05 10:20:23 PM: [PROXY -> WEBSERVICE] Response sent: QID QUERY-2

##### CLIENT 1 - output #####
client1 Request1: (retrieve (?x) (?x woman)) answerBundleSize: 3
Response1:
((?X BETTY))
((?X ALICE))
((?X EVE))
##########


##### CLIENT 2 #####
Session created..

##### CLIENT 2 - output #####
client2 Request1: (retrieve (?x) (?x woman)) answerBundleSize: 2
Response1:
((?X BETTY))
((?X ALICE))
##########

1/3/05 10:20:23 PM: [WEBSERVICE -> PROXY] Query received: [QUERY ID: 18306082,  maxBundleSize: 1 ]
1/3/05 10:20:23 PM: [PROXY] Query GET-NEXT-N-REMAINING-TUPLES: QUERY-2 SIZE: 1
1/3/05 10:20:23 PM: [PROXY -> RACER] message to racerlocalhost:8088 sent, body-Size: 42
1/3/05 10:20:23 PM: [RACER -> PROXY] message from racer localhost:8088 received

1/3/05 10:20:25 PM: Proxy Data
Query ID:                QUERY-2
Query Dailog ID:         18306082
Racer ID:                0
MaxBundle:               4

1/3/05 10:20:25 PM: [PROXY -> WEBSERVICE] Response sent: QID QUERY-2

##### CLIENT 2 - output #####
client2 Request2: (retrieve (?x) (?x woman)) answerBundleSize: 2
Response2:
((?X EVE))
((?X DORIS))
##########
```

Conclusion 6

In the conclusion, first I would like to list benefits of the enhancement done:

Benefits

This solution leads to benefits which are as follows:

- Without this enhancement owl-webservice was supporting OWL clients but now with this enhancement, it has started supporting OWL clients with the new added feature of iterative query answering.
- It has become more efficient because of the use of caching. Caching decreased the number of communication between webservice and RacerProxy and RacerServer, as you have noticed in *chapter 5 - demo execution*.

Next I would like to list challenges I have faced during the enhancement done:

Challenges

These are the challenges occurred during the project developement:

- semantic web and webservices were new fields for me to work with so it was a good experience learning new things during developement.
- I was developing with Jan and Jog, jog was working on enhansing racer proxy and we need to develop and test parallely and i also need to simulate owl-webservice which was under development and was taken cared by Jan, so it was a quite realistic situation of parallel developement and testing. I learned a lot during development from the realistic situations and also from Jan and Jog.
- the main difficult task was to understand the paper explaning owl query-answering dialogs, it was vague and not clear, it has just a bunch of ideas and no rules, it was not strict and straight, so I chose some ideas to implement in my work.

In last I would like to list some of the prospective enhancement in the current solution:

Future Enhancement

These are the prospective enhancements:

- Better comparision logic for nRQL requests: currently it is simple string comparision but it should be more logical and generalized.
- Handling of termination from both sides are not handled well in current solution. It should be taken cared well in the future.
- Caches work well, but there is no mechanism of checking of redundant results.
- Racer Proxy needed to be enhanced in a way that it is not ready to get notification from QDialog of termination of QDialog.
- Allowed idle time for sessions and query dialogs are hard coded in code, so those should be configurable from config file.
- SessionRecycler and DialogRecycler threads are running even if there is no active session or dialog, those can be turned off in the current code but are not configurable from config file.

References

- About the DAML Language
  http://www.daml.org/about.html
- World wide web consortium issues RDF and OWL recommendations
  http://www.w3.org/2004/01/sws-pressrelease
- racer-system products
  http://www.racer-systems.com/products/index.phtml
- The semantic web: an introduction
  http://infomesh.net/2001/swintro/
- What is an ontology?
  http://www-ksl.stanford.edu/kst/what-is-an-ontology.html
- Inference engines for the semantic web
  http://www.semanticweb.org/inference.html
- The RACER system download page
  http://www.sts.tu-harburg.de/~r.f.moeller/racer/download.html
- OWL-QL – A language for deductive query answering on the semantic web
  http://ksl-web.stanford.edu/KSL_Abstracts/KSL-03-14.html
- Web Services
  http://www-106.ibm.com/developerworks/webservices/library/w-ovr/#h2
- Introduction to XMLBeans
  http://xmlbeans.apache.org/