



Reengineering of the RACER Proxy

Iterative Query Answering

Jog Ratna Maharjan

Submitted in partial fulfillment of the requirements for the degree
Master of Science in Information and Media Technologies

Supervised by
Prof. Dr. Ralf Möller
Atila Kaya

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

Hamburg, February 2005

ABSTRACT

In an attempt to improve the existing system with more functionality and increased efficiency, the concept of RACER proxy was developed, which further evolved as the RACER system itself advanced.

The need for the system to accept multiple connections simultaneously from the multi-platform clients by means of heterogeneous message exchange protocols (e.g. SOAP) was one of the reasons for reengineering of the RACER proxy system. But more importantly, it was the evolution of RACER system itself with advance query answering feature that stimulated the project of reengineering the proxy system.

The whole reengineering process was carried out following standard incremental software engineering paradigm, from analysis to final testing and this document is the reflection of the whole work process in the same sequential order.

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, Feb 2005
Jog Ratna Maharjan

ACKNOWLEDGEMENT

I would like to take this opportunity to thank my project supervisor **Prof. Dr. Ralf Möller**, *Technical University of Hamburg-Harburg*, for his unique way of inspiring students through clarity of thought, enthusiasm and caring. His technical excellence on the subject, unwavering faith and constant encouragement were very helpful to complete this project successfully.

Atila Kaya, my project advisor, is due a special note of thanks for introducing me to the RACER, Proxy systems and for all his guidance and support throughout the project. This project work was enabled and sustained by his vision and ideas.

I would like to acknowledge **Christian Finckler**, *Fachhochschule Wedel* the developer of primary RACER proxy, for providing the basic infrastructure to start the project.

I wish to extend my thanks to my colleague **Tejas Doshi**, for working side by side from analysis till the final presentation.

I wish to thank **Jan Galinski** for his comments and ideas on the various topics during development and finally packing individual project components into single system.

Finally, thank you to all of my professors and friends for their contribution to this work.

CONTENTS

ABSTRACT	i
DECLARATION	ii
ACKNOWLEDGEMENT	iii
CONTENTS	iv
LIST OF FIGURES.....	v
Introduction	1
RACER.....	1
nRQL.....	3
RACER Client	3
RACER Proxy	4
Motivation	6
Lehman’s Laws [LB 1985].....	6
Resource Overload Propagation	6
1. Iterative Query Answering	7
2. Message Interchange Interface (Web services).....	8
Reengineering	10
Analysis	10
<i>Multi-session queries</i>	<i>11</i>
<i>Server unavailability.....</i>	<i>11</i>
<i>Standard Interfaces.....</i>	<i>12</i>
Design	14
<i>Parameter Logging.....</i>	<i>14</i>
<i>Architecture</i>	<i>16</i>
<i>Components</i>	<i>17</i>
Demonstration	21
Scenario.....	21
Execution Process.....	22
<i>Query 1</i>	<i>22</i>
<i>Query 2</i>	<i>24</i>
<i>Query 3</i>	<i>24</i>
Conclusion & Outlook.....	26
Features.....	26
Further More	27
REFERENCE	28
Literature & WWW Addresses.....	28
Related Projects.....	28
Software tools.....	28
APPENDIX	29
A. “Family.racer” knowledge base file (TBOX & ABOX).....	29
B. Complete output of Demo	30
C. Interface between RACER client and RACER proxy	32

LIST OF FIGURES

Fig 1.1 RACER client server query model	1
Fig 1.2 Concept hierarchy for the “family” TBox [RACER Manual 1.7.19 2004].....	2
Fig 1.3 Depiction of the” ABox smith-family”. [RACER Manual 1.7.19 2004].....	2
Fig 1.4 RICE showing both the query and result.	4
Fig 1.5 Relaying of requests & responses with introduction of RACER Proxy	4
Fig 1.6 RACER proxy routing requests from multiple clients & re-routing responses from multiple servers	5
Fig 2.1 RACER & client interaction in" tuple-at-a-time" mode	7
Fig 2.2 RACER proxy message interchange interfaces.....	9
Fig 3.1 RACER, Proxy & client interaction in” tuple-at-a-time” mode	10
Fig 3.2 Message interchange interfaces between client, proxy and the RACER.....	13
Fig 3.3 RACER, Proxy & client interaction with storage	15
Fig 3.4 New RACER Proxy Architecture	16
Fig 4.1 Interaction between Client, Proxy and RACER server	21
Fig APPNEDIX C. Interface between RACER Client and the RACER Proxy	32

Chapter 1

Introduction

A typical RACER query interaction could be visualized as a simple two tier client/server model constituted by a *RACER server* and a *RACER client*. Requests and responses between them are being transferred as $(n)RQL$ statements.

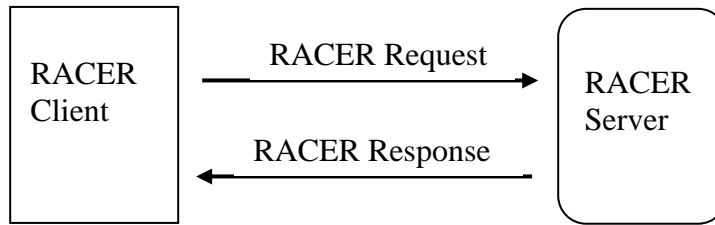


Fig 1.1 RACER client server query model

In above figure Fig 1.1, the client prepares a typical nRQL (*pronounced nercle*) query, analogous to an SQL query, to be sent to the server. The server holds the knowledge base for answering that query. The knowledge base stored on the server can be transferred to the server prior to the query or even along with the query itself. Upon arrival of the query on the server, the server computes the results and returns the response back to the client once again as same nRQL statement.

Before getting in-depth details regarding this project work, let us get familiar with the major components of this interaction system in this chapter. Following sections in this chapter provides brief overview of these aforementioned terminologies as well as some more that will be used in later chapters.

RACER

RACER (**R**enamed **A**Box and **C**oncept **E**xpression **R**easoner) system was developed by Prof. Dr.Ralf Möller and Volker Haarslev in 1999 at University of Hamburg, Germany. Since then it is being used in many research projects “*as a knowledge representation system that implements a highly optimized tableau calculus for very expressive description logic*”. [RACER Manual 1.7.19 2004].

RACER system provides reasoning for many *TBoxes* and *ABoxes*. A collection of concept axioms is called a **TBox** (**T**erminological **B**ox) and a collection of assertional axioms is called an **ABox** (**A**ssertional **B**ox). [for details ref. A. “*Family.racer*” *knowledge base file (TBOX & ABOX), APPENDIX*]

Given a TBox, different queries can be answered. Based on the logical semantics of the representation language, different kinds of queries are defined as inference problems.

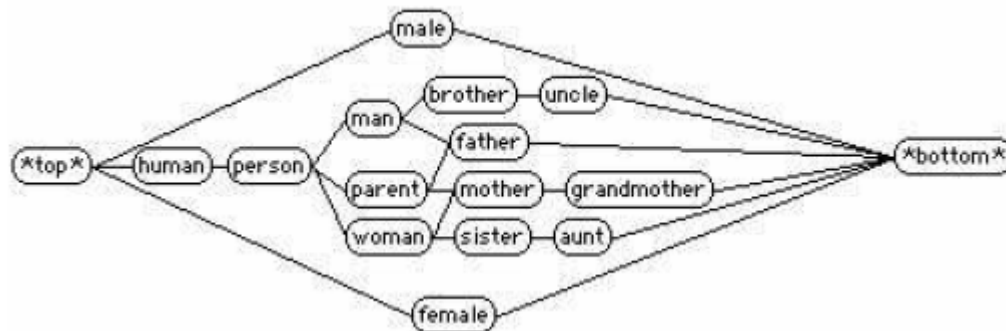


Fig 1.2 Concept hierarchy for the “family” TBox [RACER Manual 1.7.19 2004]

Some of the possible queries are;

- Concept consistency w.r.t. a TBox: i.e. is the set of objects described by a concept empty?
- Concept subsumption w.r.t. a TBox: Is there a subset relationship between the set of objects described by two concepts?

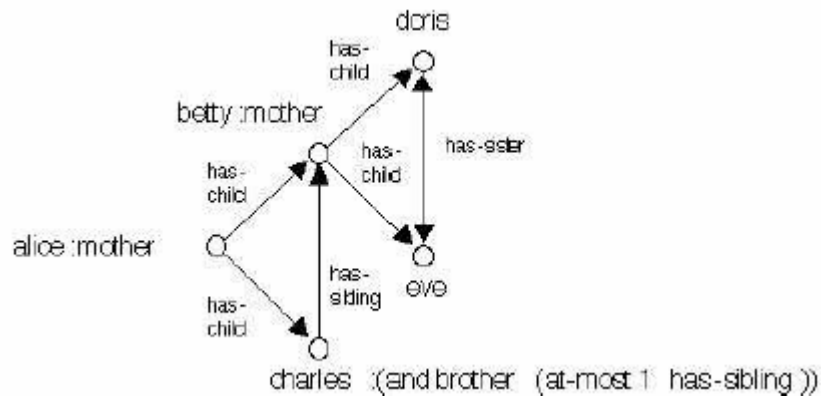


Fig 1.3 Depiction of the” ABox smith-family”. [RACER Manual 1.7.19 2004]

If also an ABox is given, among others, further more types of queries are possible like;

- Check the consistency of an ABox w.r.t. a TBox: Are the restrictions given in an ABox w.r.t. a TBox too strong, i.e., do they contradict each other? Other queries are only possible w.r.t. consistent ABoxes.
- Instance retrieval w.r.t. an ABox and a TBox: Find all individuals from an ABox such that the objects they stand for can be proven to be a member of a set of objects described by a certain query concept.

nRQL

nRQL (new RACER Query Language) is the language of RACER for message interchange. It is derived from the previous standard RACER Query Language (RQL). Based on the complexity of the query syntax, a nRQL queries can be classified as unary, binary atoms queries or the complex queries.

A typical simple nRQL request query, which a client in above Fig 1.1 could have send looks like;

```
(retrieve (?x) (?x woman))
```

This query asks the server to return all the instance of woman from the “ABox” located on RACER Server. For which the RACER server loaded with “family knowledge base” would reply back another nRQL statement with variable value binding list as follows;

```
((?x EVE)) ((?x DORIS)) ((?x ALICE)) ((?x BETTY))
```

It is not just the complexity of the query which is used to categorize the queries. nRQL statements can be further categorized as *Statements* and *Queries* depending upon the state of knowledge base after its execution.

Statements are those nRQL statements which can change the stored knowledge base (ABOX or TBOX) after its execution, analogous to the “UPDATE” statements in SQL. Whereas queries are those nRQL statements which don’t alter the internal knowledge base structure (ABOX or TBOX) stored in the RACER server after its execution, analogous to “SELECT” statements in SQL.

In this project we mainly focus more on the queries than the statements, mainly unary atom queries and few complex queries with two atoms.

RACER Client

RACER client is a multi-platform (both hardware and operating system) system which can make socket connection (basically TCP or HTTP) to server over any network protocols supported by the RACER server or by the intermediary systems between the server and the client.

There already exist some client implementations which provide the interface to send and receive nRQL queries to/from the RACER server respectively. Some of them are simple command prompt based interfaces like DIG client, whereas some have extensive GUI interface even showing detail hierarchical knowledge base structure stored on the server.

RICE (RACER Interactive Client Environment) is one of such simple java based GUI RACER client with extensive features. It not only provides facility to send (n)RQL queries to the RACER server directly (or through the RACER proxy), but also shows a list of “TBoxes” and the desired “ABoxes” of the loaded knowledge base.

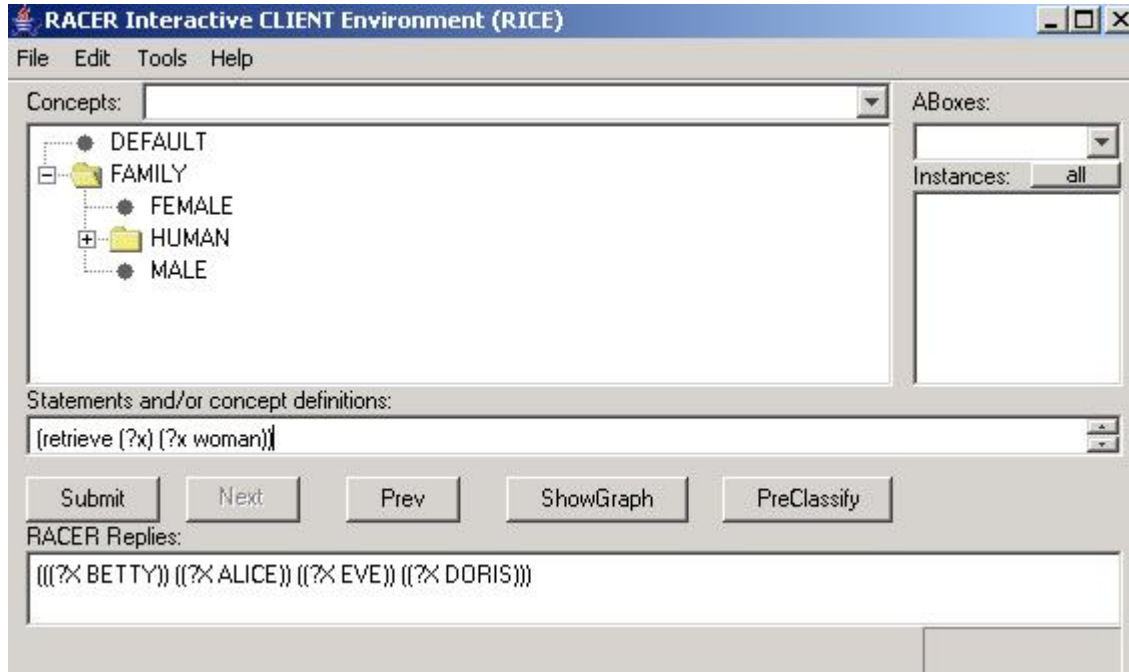


Fig 1.4 RICE showing both the query and result.

RACER Proxy

The basic objective of the development of RACER proxy was similar to most other proxy systems. Like a web or mail proxy, the RACER proxy was primarily developed to relay the messages to and forth between RACER server and client.

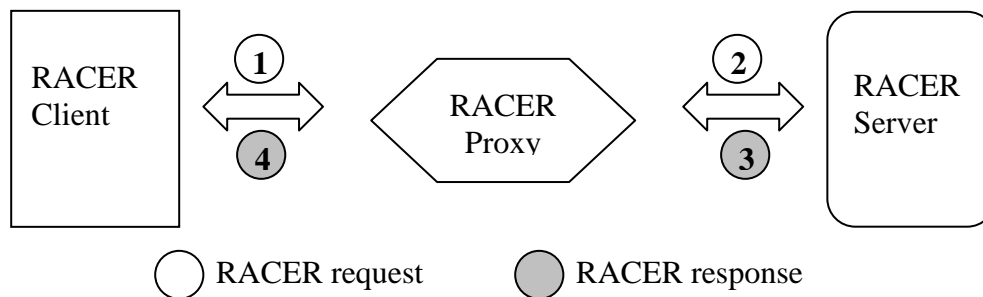


Fig 1.5 Relaying of requests & responses with introduction of RACER Proxy

The primary proxy (or the middleware) takes the RACER queries from one or more RACER clients as input then forwards those queries one by one to the RACER server. The server in turn processes those incoming query and sends back the generated results

to the proxy without having to care about the end client platform or even the message exchange protocol that the end client is using to send the request at the first place. Some of the overload on the server side regarding clients is reduced even with this simple message-relay proxy.

As any other proxies, the primary RACER proxy was able to handle multiple client requests simultaneously. Although it didn't use any dynamic message processing system for the en-queued messages (which has not yet been processed), it used simple FIFO queue to store those newly arrived requests. Once the RACER server became free, the proxy controller used to read and remove these requests from the queue for further processing. This was one the immediate benefit of having RACER proxy over the normal client/server system, where there was limitation of handling only one client request at a time.

The proxy provided support for the multiple connections not only on the client side but also supported multiple RACER servers loaded with same knowledge base. The advent of the proxy system with the support of multiple servers provided load safe redundant backup for the RACER server. If one of the RACER servers failed or became busy, while arrival of the new request, then the request could be easily routed to another available server.

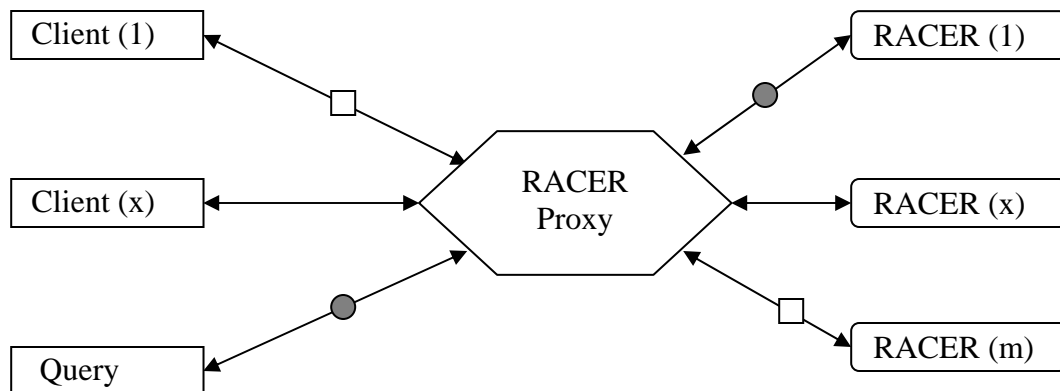


Fig 1.6 RACER proxy routing requests from multiple clients & re-routing responses from multiple servers

RACER proxy was able to provide primitive load sharing functionality, by routing the queries to the first available RACER server. But in the case of statements it still sent an incoming statement to all RACER servers, so that they all have the same state after processing the statement.

This feature in turn increases the efficiency of the whole RACER interaction model.

Chapter 2

Motivation

The demand of more and more features in an existing system and further implications introduced by the implementation of those features in the system are two major factors identified by Lehman & Balady (1985) for the reengineering of any software.

Lehman's Laws [LB 1985]

1. Continuing change

"A program that is used in a real-world environment must change, or become progressively less useful in that environment."

2. Increasing complexity

"As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure."

Not being far from this software evolution theory, Reengineering of the RACER proxy was also stimulated by following two factors;

- 1. Need for the support of the RACER server's new "Iterative Query Answering" feature.*
- 2. Need to incorporate new web service module for message interchange.*

Resource Overload Propagation

The previous version of RACER 1.7.x suffered a problem of generating complete result set for every query it received in a single execution. It not just used heavy resource of the RACER system while processing huge result set, but the heavy resource usage was propagated to the proxy system also.

Frequent queries with a huge result sets could exceed the threshold resource of the proxy, even RACER system itself and finally cease the system.

The heavy resources usage was not just within the system; this bulk result set caused heavy network congestion between the client and the server while transferring such huge result sets.

Much worse scenario would be;

What if the client's requirement was just a partial subset (maybe first 10) out of the huge result set(10 thousand) that was generated by the server?

The generation of such huge result at the first place was resource overhead on the server side, but the transfer of such huge result till proxy and relaying back to client would be much more overload on the whole interaction path of the system.

Devoid of such feature in the system, makes system less useful and maybe one day might be totally abandoned, that is what Lehman’s first law of “Continuing Change” describes.

1. Iterative Query Answering

In order to alleviate the above problem of *resource overhead propagation*, and following “Continuing Change” Law to make the RACER more useful, RACER server introduced the feature to generate and return partial results from the server side itself. Out of many querying features introduced in version 1.8, “*tuple-at-a-time*” query feature allows any RACER client to get partial results at a time.

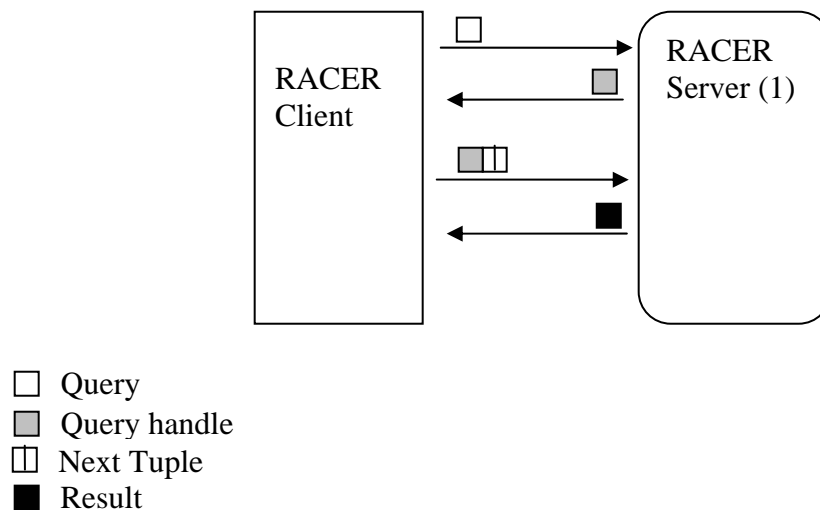


Fig 2.1 RACER & client interaction in” *tuple-at-a-time*” mode

1. RACER client sends some nRQL query request to the RACER server running at *tuple-at-a-time* mode.
2. RACER server returns a unique (for that particular server) query handle for that RACER query as a response to the client.
3. The client sends request to get first tuple along with the query handle that it got from the RACER in previous step.
4. In return, the RACER will return the first tuple from the generated result set for that query.

5. If the client requires more result tuples, it can continuously send *get-next-tuple* / *get-all-remaining-tuple* requests to the server.
This kind of repetitively delivering the result, tuple by tuple or as tuple bundle, as requested is called *iterative query answering*.

The introduction of “*tuple-at-a-time*” feature alleviated the problem to some extent, to get the partial result set directly from the server. But still, each of the clients running on multiplatform will have to be remodeled to adapt this new feature. The query and re-query format and result set retrieval specification changes will have to be adapted by each of the clients, which were developed in different languages and was working on multiple platforms. This type of redesigning of the client for each new feature that is being added or will be added in future could be very cumbersome and unfeasible in long run. There would be great potential of heterogeneous RACER clients, some working on previous version specifications and some implementing even future features at the same time. The synchronization of these client’s updates would be still more troublesome once RACER enters into commercial market scenario with multiple commercial client applications. This was what Lehman’s second law of “Increasing complexity” forecasted.

In order to simplify the complexity, the adoption of this new “iterative query answering feature” was shifted to the RACER proxy system. That would not only reduce the cumbersome of updating all the RACER clients widely distributed but also reduce both the client and the server side processing load with efficient centralized processing.

2. Message Interchange Interface (Web services)

Prior to redesign, RACER proxy was accessible to RACER clients by means of socket connections only, either using TCP or HTTP connection protocol. As an enhancement in RACER proxy, there was a concurrent development of web services module going on to support the OWL-QL [*RACER OWL-QL Interface, 2005*]. OWL-QL (Web Ontology Language – Query Language) is not only an xml based query language for the semantic web but also a protocol describing query-answer dialogs.

In order to incorporate particular section of that module called RACER Proxy web service interface [*RACER Proxy WS Interface, 2005*] on the same proxy system, the proxy needed to provide much simpler and efficient medium for the interchanging messages.

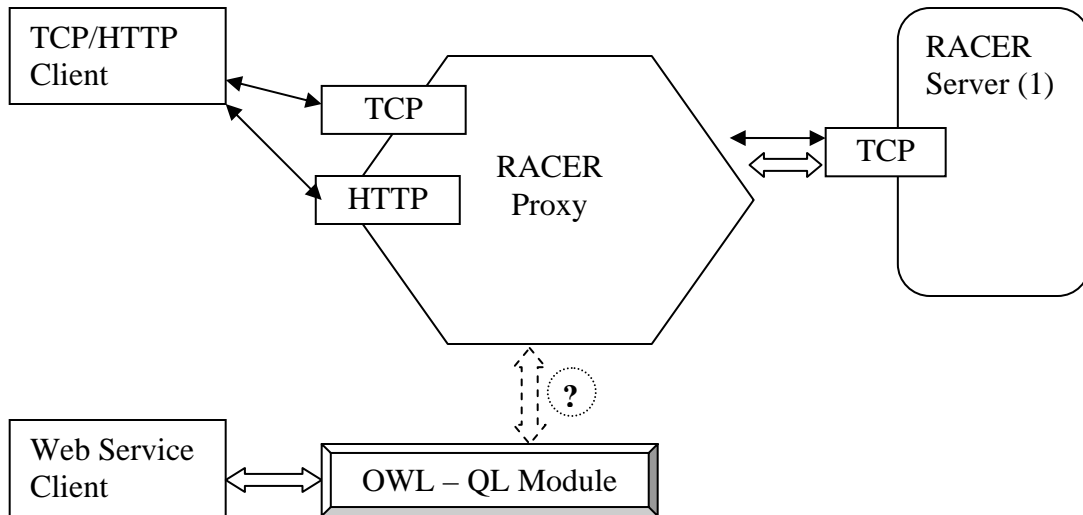


Fig 2.2 RACER proxy message interchange interfaces

So, on analyzing Fig 2.2, the need of redesigning of an interface was not just limited between RACER server and the proxy, but it required some standard interface between RACER clients and the proxy for actually using this new query answering feature.

Considering above two requirements to be fulfilled, the my project can be defined as

“Reengineering of the RACER proxy to support iterative query answering for clients using Web Service”.*

* In coordination with [RACER OWL-QL Interface, 2005] & [RACER Proxy WS Interface, 2005] projects.

Chapter 3

Reengineering

This chapter provides details regarding analysis to final implementation and testing of the reengineered proxy. The formation of requirements and possible solutions were determined during the analysis phase. Based on the outcome of the analysis of the initial system, the architecture design was developed during the design phase. The implementation phase was just the realization of the architectural design in JAVA.

Analysis

The introduction of the proxy in the “iterative query answering” model (*ref Fig 2.1 RACER & client interaction in “tuple-at-a-time” mode*) changed the whole interaction scenario.

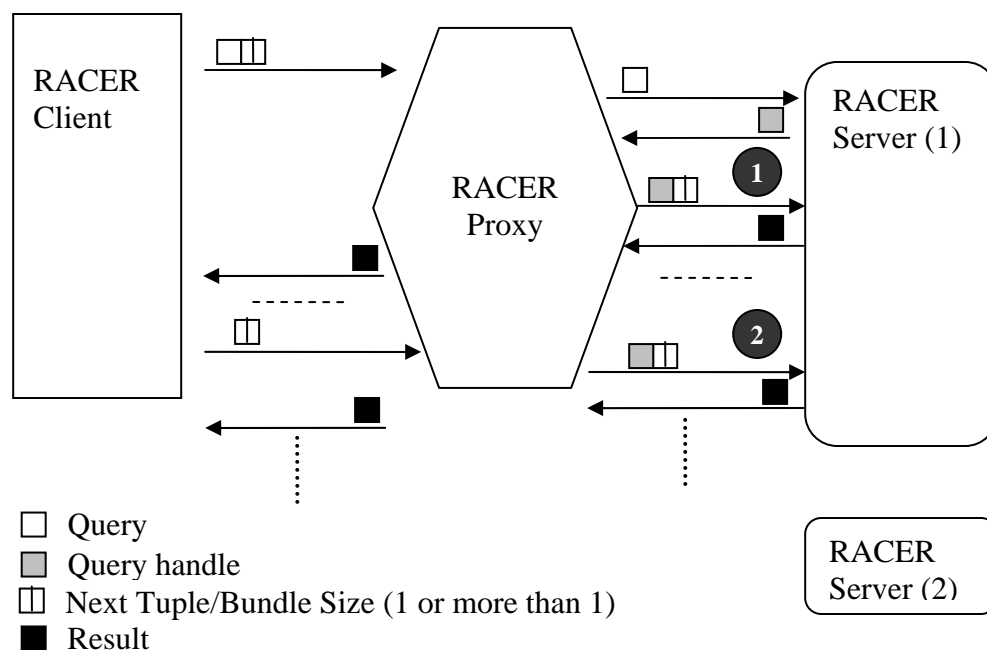


Fig 3.1 RACER, Proxy & client interaction in “tuple-at-a-time” mode

The introduction of proxy in between the clients and the server reduced the interaction between a client and the server by transferring all the iterative interaction responsibilities to the proxy. The simple request/response scenario changed, once proxy had to handle multiple client requests and multiple RACER’s in the iterative query answering model.

Following initial requirements have been determined during top level analysis.

Requirements

- 1) Identification of the queries and client sessions
- 2) Request query routing during multi-session query.
- 3) Re-routing strategy in case of server unavailability
- 4) Standard interface medium between client & the proxy for message interchanges.

Multi-session queries

The previous scenario of relaying the message to and from the proxy system within a single session no more existed, once iterative query answering feature was introduced. As shown in the figure Fig 3.1, there can be certain period of delay between the first iterative query request and the second one. So, same client can participate in multiple query sessions.

Due to this multi-session interaction nature, routing of the incoming query requests from the client needs to be handled carefully. The proxy could no longer route the incoming requests (queries) to any of the available RACER server as before (*ref. Fig 1.6*). The proxy must first determine whether it is a normal query or an iterative query request. If the request is an iterative one, then it has to route to the same RACER system, which processed its initial query.

Server unavailability

Continuing with the iterative query, what could be the consequences if the desired RACER server becomes unavailable during the second iterative query in the *Fig 3.1*. The most likely solution to deal with such probable situation by the proxy can be;

- (a) Waiting long enough for the RACER server to become available
- (b) Or sending the iterative query to next available server, without any delay.

The first option (a) seems a simple solution without much overhead, but in the worst case it could lead to the deadlock situation if the desired server never becomes available. In a scenario where the desired server breaks down or needs to process a long running query resulting very large result set, then the query will never be answered.

The second option (b) seems better than the first (a) because it doesn't arise the deadlock problem, unless all of RACER servers break down or remains busy forever. The option can be favored much more because of the efficient response time, as there isn't any delay even if the desired server is found busy. Finally, as the query can be immediately routed to the next available server, it provides better resource utilization and good load sharing.

Similar to the prior option, this option is also not devoid of few overhead.

Even though all RACER servers are loaded with same knowledge base, not all of them can respond to the subsequent iterative query. None of the RACER servers, except the desired and unfortunately unavailable server, can give the correct result corresponding to the query handle passed as quest query from proxy to the server. The query handle (i.e. some unique name) that was generated by the desired server during the first query is never propagated to other servers, which made other servers unaware of the mapping of certain query handle to that query.

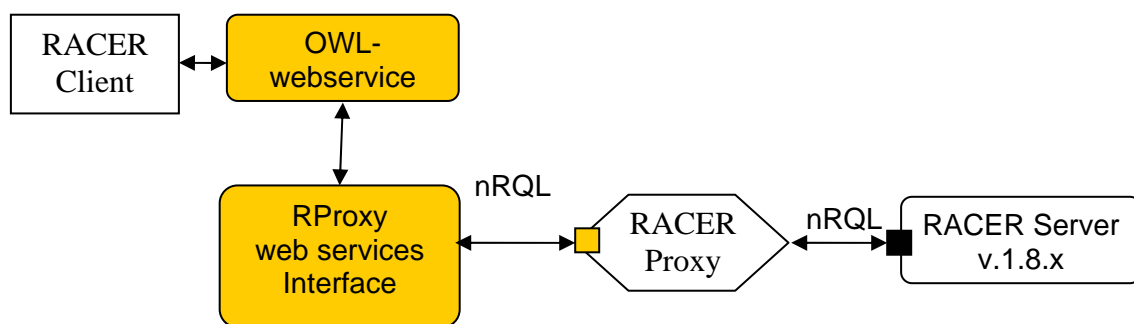
It is also not compulsory that all of the servers will generate same query handle (like “QUERY – ID“, with unique ID) for any particular query that is send concurrently to all RACER servers.

So the solution (b) which at the beginning seemed very impressive could only be used at the cost of heavy overhead of resources. In case of unavailability of the desired server, the proxy would have to send initial query once again to new RACER server in order to get the new query handle, and then only could retrieve desired results.

Standard Interfaces

The interfacing medium between the client and the proxy as well as between the proxy and the server needs to be defined precisely. For the latter part, new RACER’s API [RACER Query 1.8, 2004] for iterative query retrieval, serves as standard for the interface. Standard nRQL statements and queries based on the new server API can be sent over TCP socket connection to the RACER server.

Whereas between the client (web service) and the proxy, there existed many components as intermediaries, that abstracted the client query request and even the client connection itself. Following figure Fig 3.2 depicts two of such major components that can exist between the client and the RACER Proxy. [For details ref C. Interface between RACER client and RACER proxy, APPENDIX].



- Intermediary components between client and the RACER Proxy
- Racer proxy interface for application level call from “Rproxy WS Interface”
- TCP socket interface for the proxy to connect to RACER

Fig 3.2 Message interchange interfaces between client, proxy and the RACER

The connection between “RProxy web services Interface” & RACER Proxy in above figure Fig 3.2 could be implemented as one of followings;

- (a) Socket connection (TCP)
- (b) Remote Procedure Call (RPC)
- (c) Application level function call

At first glance, options TCP & RPC, both seems nice as it provides flexibility to run those intermediary components apart from the proxy, in distributed environment. But if those intermediary components are be to finally coordinated into single package along with the proxy, then simple application level functional call would result in a much efficient and secure message interchange interface.

Design

With through analysis of existing primary proxy system and considering the pros and the cons of the possible alternatives for implementing new features on the system, the architecture of the previous RACER proxy system was redesigned to the some extent.

Parameter Logging

For handling new multi-session querying feature by the proxy, proxy needs some storage medium to log information regarding this multi-session query.

The information that the proxy needs to keep track related with the query are as follows;

Parameters	Description
QDID	<p>Query Dialog ID, a unique numerical value provided by the “RACER proxy web service interface” [<i>C. Interface between RACER client and RACER proxy</i>], which acts as the client connection for each request from that interface.</p> <p>QDID is pass along the nRQL query during first request and again attached along with the bundle size in every subsequent request related to that query.</p>
RID	RACER ID , a unique numerical index value corresponding to the RACER server that processed the given query.
QHID	Query Handler ID , a unique value returned by the RACER server 1.8.x (running on tuple-at-a-time mode) on the initial retrieval query.

The logging of above parameters for every new query that arrives at the proxy would help the proxy to re-route further iterative calls related with that query to correct RACER server. The previous scenario of figure Fig. 3.1 is modified with the proxy storage table and QDID in Fig. 3.3.

The previous analysis scenario of server unavailability still cannot be fulfilled with just above parameters. In case of the server unavailability, after allowed number of retries the request must be fulfilled by next available server. In order start this query re-routing to new server from the beginning, the proxy requires still two more parameters as shown in the table below;

Parameters	Description
Query	nRQL , initial Query
MBS	MaxBundleSize , the cumulative number of results returned till date for that particular QDID, or the sum total of the bundle size request till date.

With the original nRQL query stored, the proxy can route the initial query to the next available server from the start.

With MaxBundleSize parameter, which actually stores the cumulative result Bundle size delivered till now for that query, provides an offset to make a request query “the *get-next-n-remaining-tuple*” during the server unavailability.

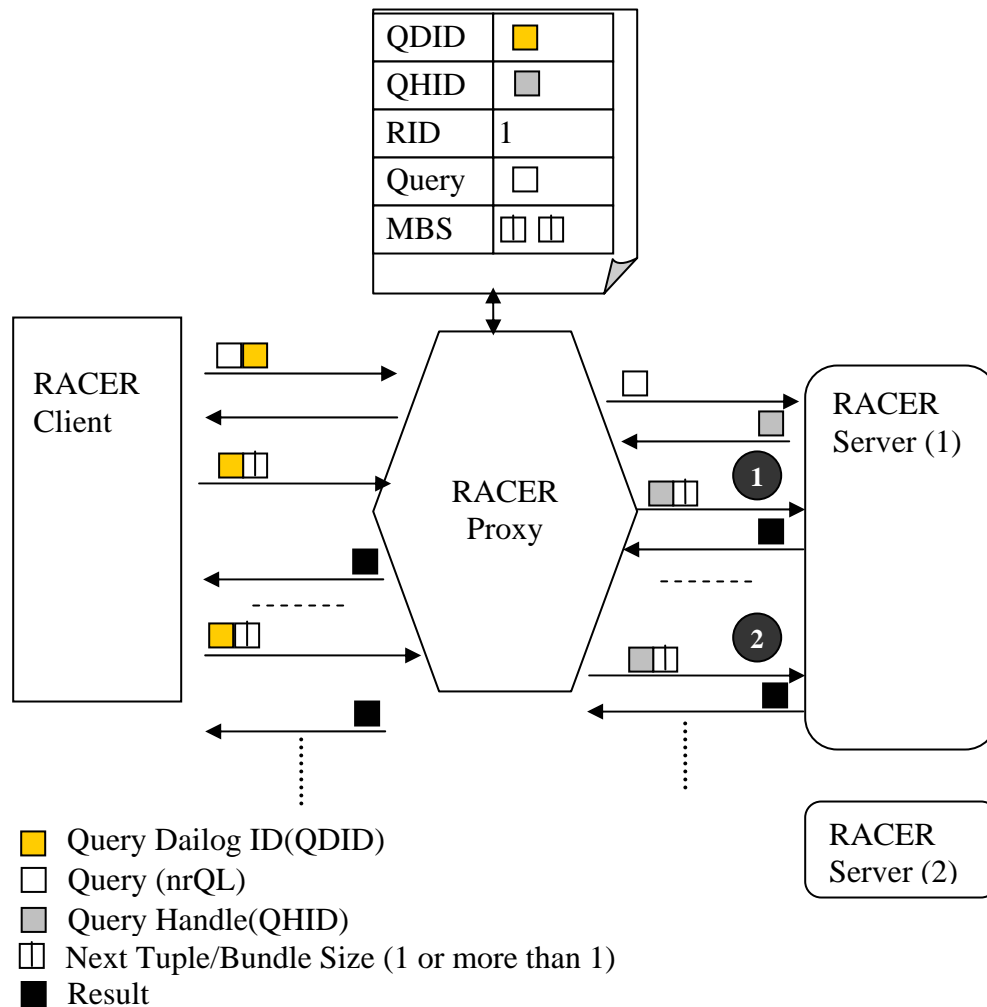


Fig 3.3 RACER, Proxy & client interaction with storage

Architecture

Following figure highlights the significant functional components of the reengineered proxy.

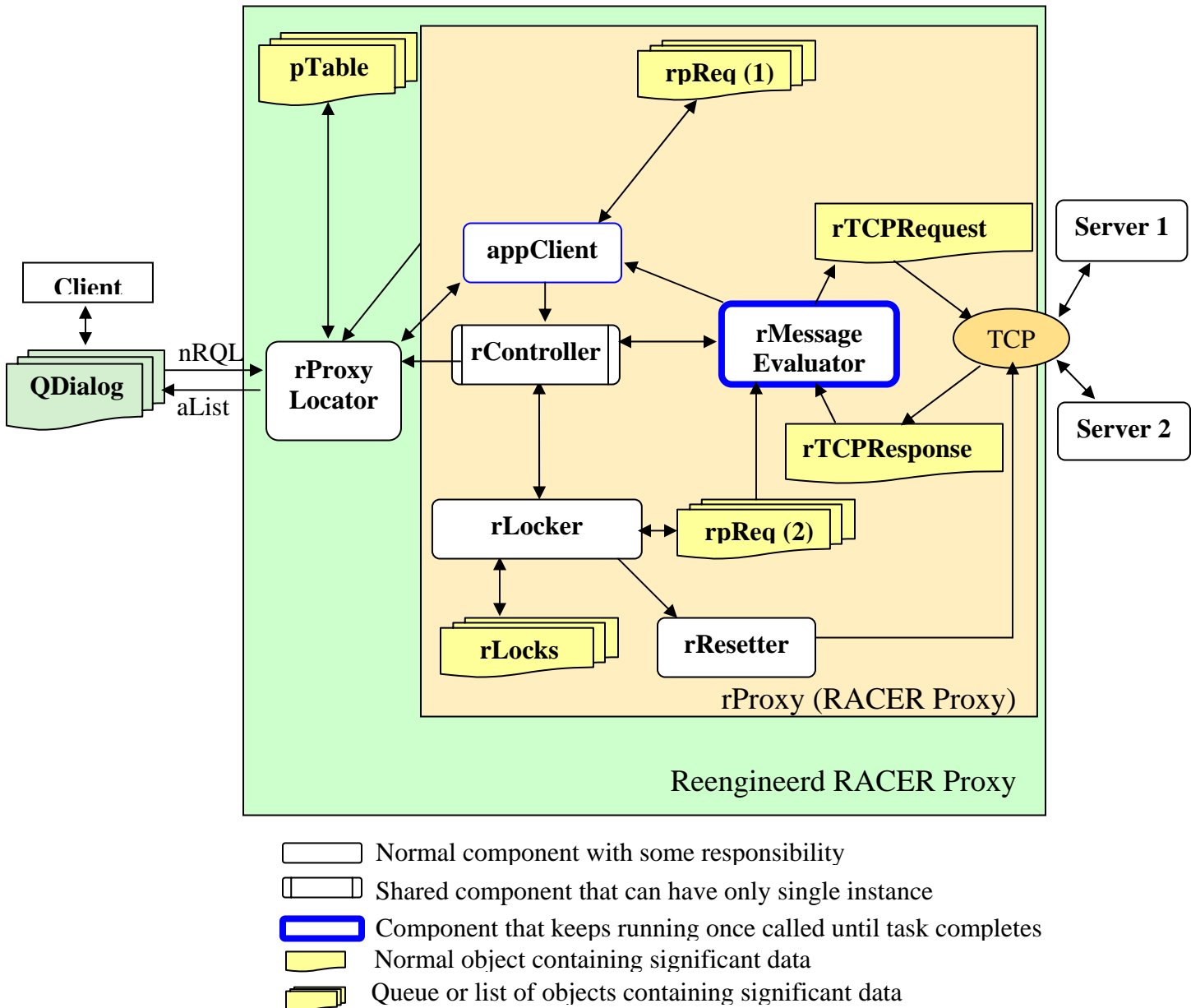


Fig 3.4 New RACER Proxy Architecture

Most of the functional components in the reengineered proxy were basically derived from the existing ones with basic modification to meet the requirements determined in the analysis phase in the previous section.

Components

1. rProxy (RACER Proxy): RACER Proxy component is the main proxy server, which wraps all the major components of the proxy system. It invokes *rController (RACER Controller)* which is responsible to setup the connection with the RACER server based on the configuration parameters provided to it. It also starts port listener thread to accept incoming TCP/HTTP request on the designated ports provided by the configuration.

2. rProxyLocator (RACER Proxy Locator): It's a locator component responsible to start the RACER Proxy component (*rProxy*), if it is not already started. It provides an interface for "QDailog" (Racer Proxy Web Service Interface) [*for details see, C. Interface between RACER client and RACER proxy, APPENDIX*] to call application level functions with the help of *appClient (Application Client Interface)*.

In the case of iterative query, it reads the parameter values from the *cache* queue which stores *pTable (Proxy Table)* data corresponding to that iterative query. From the response message included within *rpReq (RACER Proxy Request)* it enters or updates following query parameter *pTable* data in the *cache*;

QDID (*initial or server unavailable*),

QHID (*initial or server unavailable*),

RID (*initial or server unavailable*)

Query (*during first query*),

maxBundleSize (*initial or iterative*).

3. appClient (Application Client Interface): It's an application connection interface component, with utility functions to send queries to the server. In the case of simultaneous application calls from many clients, it stores the *rpReq (RACER Proxy Request)* into its local queue (*rpReq (1) Queue*), which will be later forwarded to *rController (RACER Controller)*.

4. rController (RACER Controller): This component is responsible for controlling connections from the proxy to the RACER server. It invokes a thread *rMessageEvaluator (RACER Message Evaluator)* which is responsible for further message processing, and in meantime relays the rest message from the *appClient* to *rLocker (RACER Locker)* to find the available free RACER server.

5. rLocker (RACER Locker): Checks the first available RACER and makes it busy by setting value of the index corresponding to that RACER in a vector *rLocks (RACER Locks)* as false.

In the case of an iterative query, it checks status of desired RACER server and locks if available, else waits for *100 milliseconds* and retries for at most 3 times. If it finds that the server is still not available, then it invokes *rResetter (RACER Query Resetter)* component.

Once locking phase is completed, then it stores the *rpReq* into the *rpReq(2) Queue* for further processing.

6. pTable (Proxy Table): pTable (or query information storage table as described in *parameter logging* section before) holds all the necessary parameters regarding the query session.

7. rResetter (RACER Resetter): This component is responsible to make request to next free RACER server, if the desired RACER is found busy for long time. It gets all parameters (Query,Maxbundlsize) from the *rpRequest*.

8. rMessageEvaluator (RACER Message Evaluator): This component is continuously running, to read *rpRequest(2)* queue for latest *rpRequest*. It forwards the *rpRequest* as *rTCPRequest* message over TCP connection to the desired RACER server. Once response message is received from RACER, it calls *rController* to unlock the RACER.

9. rpReq (RACER Proxy Request): This is the main data component (message component), routed among the proxy components, holding all the significant query and response parameters related with that particular query. Both *rTCPRequest* and *rTCPResponse* messages are derived from this component.

Implementation

All of the aforementioned components in the design phase were converted into the JAVA classes in the implementation phase. Following are the list of major classes that were developed or modified to meet the new requirements.

	Class Name	Purpose	Access/Type
1	RacerProxy	The main Proxy class, which wraps all other classes	Public Singleton class
2	RacerProxyLocator	- Starts proxy (if not started) - acts as interface for application connection. - updates and adds <i>ProxyTable (pTable)</i> data into <i>Cache (HashMap)</i> .	Public
3	AppClient	- Locally queues incoming requests - Provides utility functions for querying	Public
4	RacerController	Sets up connection with the RACER server	Public Singleton class
5	RacerLocker	Contains synchronized functions which; - Locks the message processing RACER server by setting boolean value on <i>rLocks (Array)</i> . - & Unlocks it after arrival of response.	Public
6	ProxyTable	- Holds query request parameter values	Public
7	Racer Resetter	Contains single <i>static</i> function to query next available server, if the desired server becomes unavailable	Public
8	RacerMessageEvaluator	- Processes the request messages (<i>rpRequest</i>) available in the <i>ArrayList</i> . - sends/gets request/responses to/from RACER with TCP connection	Public extends Thread
9	RacerProxyRequest	Encapsulates all query as well as response related data and serves as basic message interchange object within the whole proxy.	Public

Apart from these major classes many utility classes were also created for formatting RACER messages.

Testing

Unit testing and integration testing was performed using predefined nRQL queries. The proxy system supported both unary and binary atom queries ranging from simple to complex query types.

1. Unary query:

```
(retrieve (?x) (?x woman)) [BUNDLE SIZE = 2]
```

2. Binary complex query:

```
(retrieve (?x ?y) (?x ?y has-father)) [BUNDLE SIZE = 2]
```

Unit testing was carried out using above mentioned nRQL queries directly instead of its OWL counterparts. Test class was created with above mentioned queries.

Integration testing was performed in coordination with the RACER Proxy Web Service Interface [*RACER Proxy WS Interface, 2005*] component.

The absence of RACER OWL-QL Interface [*RACER OWL-QL Interface, 2005*] was fulfilled by a dummy translator class which returned above mentioned nRQL queries to RACER Proxy Web Service Interface.

Both tests resulted into following desired outputs;

1. Unary query:

```
((?x EVE))  
((?x DORIS))
```

2. Binary query

```
(( (?X EVE) (?Y CHARLES) ) )  
NIL
```

Details of the execution and processing of one of the test cases is presented in the following chapter named **Demonstration**.

Chapter 4

Demonstration

This demo execution is done in coordination with the “RACER Proxy Web Service Interface” [RACER Proxy WS Interface, 2005] and simulated “RACER OWL-QL Interface” [RACER OWL-QL Interface, 2005]. The infamous “family knowledge base” (ref. A. “Family.racer” knowledge base, APPENDIX) has been loaded into the RACER server.

Scenario

There exist two clients, who subsequently need to retrieve number of woman entries stored in the family knowledge base stored in the RACER server.

(a) Query 1

Client 1: Get first 3 woman’s entries from the family knowledge base stored in the RACER server.

(b) Query 2

Client 2: Get first 2 woman’s entries from the family knowledge base stored in the RACER server.

(c) Query 3

Client 2: Get next 2 woman’s entries (3rd & 4th) from the family knowledge base stored in the RACER server.

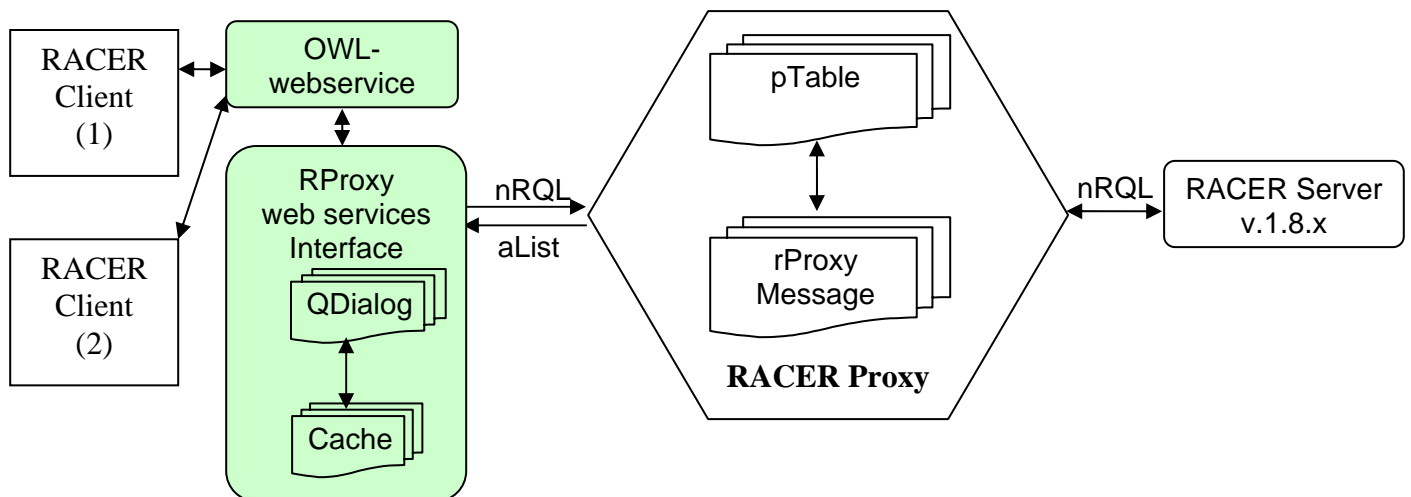


Fig 4.1 Interaction between Client, Proxy and RACER server

Execution Process

Query 1

The first RACER client (client 1) sends a request query to retrieve. The query will first be processed by the interfaces between the client and proxy, which in turn will convert the OWL query into infamous nRQL as follows;

```
(retrieve (?x) (?x woman))
```

- Then the “Racer proxy web service interface” initiates the connection with the RACER proxy. If there isn’t any instance of the RACER proxy yet, then new instance of proxy is created which starts the RACER proxy itself.

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-Messenger started  
1/3/05 10:20:11 PM: TCP-Messenger started  
.....
```

- As soon as proxy gets started it set ups the connection with the RACER server.

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  
.....
```

Once the connection between, “web service interface”, proxy and the RACER server is setup, the “web service interface” sends the above nRQL query along with the Query Dialog ID (QDID).

- Once proxy receives the nRQL statement then it checks the RACER server processing mode. If it is not running on “tuple-at-a-time” mode then it sets it to “tuple-at-a-time” mode.

OUTPUT

```
.....  
1/3/05 10:20:13 PM: [WEBSERVICE -> PROXY] Connection received  
with query: (retrieve (?x) (?x woman))  
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
```

- After this initial setup only, the proxy formats the query as nRQL request message and sends it to the RACER server, which returns back the initial nRQL response as Query Handle (QHID).

OUTPUT

```
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
```

The proxy stores this QHID, QDID, Query and the available RACER ID for future purpose into “pTable” (object).

- It then signals connection successful to the “web service interface”.

OUTPUT

```
1/3/05 10:20:19 PM: [PROXY -> WEBSERVICE] Connection successful
```

- The “web service interface” now initiates the iterative query with QDID & the bundle size to be retrieved from the server.

OUTPUT

```
1/3/05 10:20:19 PM: [WEBSERVICE -> PROXY] Query received: [QUERY  
ID: 18306082, maxBundleSize: 3 ]
```

- In the response, the proxy sends a subsequent query to get the first result from the previous server with the corresponding QHID (retrieved from the “pTable”).

OUTPUT

```
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
```

- As the requirement was 3 results, so we need to retrieve still 2 more tuples, for which proxy prepares and sends another query to the same server (after updating the “pTable” for the previous result).

OUTPUT

```
.....  
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  
.....
```

- It then updates the “pTable” and sends back the result to the “web service interface” as tuple bundles.

OUTPUT

```
.....  
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  
.....
```

Query 2

The second query requested by the client 2 requires only first two tuples to be returned. The “web service interface” had already received first three tuples before for previous identical query. So, the request can be easily fulfilled by the “web service interface” alone, by returning the two tuples from its previously cached result without any interaction with the proxy. [*RACER Proxy WS Interface, 2005*]

Query 3

The second client requires still two more results from the server, but the cache of the web service interface holds just three results out of which first two have already been delivered to this client. The web service interface cache ran short of just one more result.

- The “web service interface” sends the request for one more result to the proxy.

OUTPUT

```
1/3/05 10:20:23 PM: [WEBSERVICE -> PROXY] Query received: [QUERY ID:  
18306082, maxBundleSize: 1 ]
```

- The proxy creates and sends the request to retrieve next one tuple as requested by QDID

OUTPUT

```
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
```

- The proxy updates Ptable after it gets response from the RACER. & sends the response back to “web service interface”.

OUTPUT

```
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
```

Complete output, including client side output can be found on the APPENDIX [B.
Complete output of Demo].

.

Chapter 5

Conclusion & Outlook

Features

Apart from fulfilling the major objectives to deliver partial result sets as desired by the client relayed through proxy system to the web service client module, reengineered proxy provides some more significant features;

1. Abstraction of RACER

The change in either on the server side or on the client side was completely hidden by the RACER proxy system. The introduction of OWL-QL query handling feature was done without any intervention on the RACER system features.

No extra features (OWL-QL translator & Web Service module) had to be added on the server, which in turn decreased the server load. The proxy system acted like a bridge for the technological developments on both sides.

2. Server based caching

There was no result caching overhead on the proxy because of “*tuple-at-a-time*” processing mode on the server. The proxy don’t have to hold whole result set in the memory (primary) instead the list of references (Query Dailog ID, Query ID and the Server ID) was enough to retrieve new results from the server as the client requested.

3. General load balancing

The proxy would always route the query to the first available server, there on subsequent iterative query related to that query will be routed to the previous destined server only. But incase of “server failure” scenario if the timeout occurs then query can be rerouted to the next available server.




Further More

As RACER proxy is in its primitive stage, there is a good scope of enhancement by both addition of new features and by extending the existing ones to meet desired requirements in days to come. The introduction of the “iterative query answering” within proxy itself induces many existing features to be further enhanced. Following are few notable enhancements that are possible in the system:

- **nRQL Statements:** Handling nRQL statements from the “web service module”.
- **Inheritance:** Expanding the current “Iterative query answering” feature to both TCP and HTTP client connections.
- **Load Balancing:** Using heuristic and adaptive algorithms instead of selecting first available server during the initial query request. The cache table in reengineered proxy can be used as look up table for criteria determination.
- **Load Sharing:** Different RACER servers holding different knowledge bases and routing the request to one of the particular RACER servers which holds the desired knowledge bases.
- **Caching:** Proxy based caching to store fetched query results, so that frequently asked query results can be further delivered to also TCP and HTTP requests.

REFERENCE

Literature & WWW Addresses

-  [RACER Manual 1.7.19 2004] *RACER User's Guide and Reference Manual*
<http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-manual-1-7-19.pdf>
-  [RACER Query 1.8 2004] *RACER (v.1.8) Query Guide* *
- [Eclipse 2003] *The Java developer's guide to Eclipse*, Addison-Wesley
Sherry Shavor
-  [LB 1985] Lehman's Laws, Slide no. 6
<http://www.iam.unibe.ch/~scg/Teaching/OORPT/01Intro-long.ppt.pdf>

Related Projects

- [RACER Proxy WS Interface, 2005] *RACER Proxy Web Service Interface*,
Master's student project by Tejas Doshi in the STS Department.
- [RACER OWL-QL Interface, 2005] *Einsatz von Web-Services im Semantic Web
am Beispiel der RACER Engine und OWL-QL*.
Abschluss Diplom Arbeit, by Jan Galinski in the STS Department.

Software tools

- RACER 1.8.x Server (Windows) **, V. Haarslev, R. Möller, M. Wessel
[RACER 1.7.23] <http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-1-7-23-windows.zip>
- RICE , RACER Interactive Client Environment, Academic Medical Center, dept.
of Medical Informatics
<http://www.blg-systems.com/ronald/rice>
- RACER Proxy (Experimental), Christian Finckler, Univ. of Applied Sc. in Wedel
<http://www.sts.tu-harburg.de/~r.f.moeller/racer/RacerProxy.zip>
- J2SE 1.4, Java application development platform, Sun Microsystems
http://dlc.sun.com/jdk/j2sdk-1_4_2_07-windows-i586-p.exe
- Eclipse 3.0, open extensible IDE, Eclipse Foundation
<http://www.eclipse.org/downloads/index.php>

* *Not available publicly till date*

** *1.8.x is not available publicly till date, latest available 1.7.23*

APPENDIX

A. “Family.racer” knowledge base file (TBOX & ABOX)

```
(in-knowledge-base family smith-family)
(signature :atomic-concepts (human person female male woman man
                             parent mother father
                             grandmother aunt uncle
                             sister brother
                             only-child)
          :roles ((has-descendant :transitive t)
                  (has-child :parent has-descendant
                              :domain parent
                              :range person)
                  (has-sibling :domain (or sister brother)
                              :range (or sister brother))
                  (has-sister :parent has-sibling
                              :range (some has-gender female))
                  (has-brother :parent has-sibling
                              :range (some has-gender male))
                  (has-gender :feature t))
          :individuals (alice betty charles doris eve))

(implies person (and human (some has-gender (or female male))))
(disjoint female male)
(implies woman (and person (some has-gender female)))
(implies man (and person (some has-gender male)))

(equivalent parent (and person (some has-child person)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))

(equivalent grandmother
 (and mother
  (some has-child
   (some has-child person))))
(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))
(equivalent brother (and man (some has-sibling person)))
(equivalent sister (and woman (some has-sibling person)))

(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)

(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)

(instance charles brother)
(related charles betty has-sibling)

(related doris eve has-sister)
(related eve doris has-sister)
```

B. Complete output of Demo

```
##### 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-Messenger started
1/3/05 10:20:11 PM: TCP-Messenger 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 Dialog ID: 18306082
Racer ID: 0
MaxBundle: 3
```

Reengineering of RACER Proxy for Iterative Query Answering

```
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))
#####
```

C. Interface between RACER client and RACER proxy

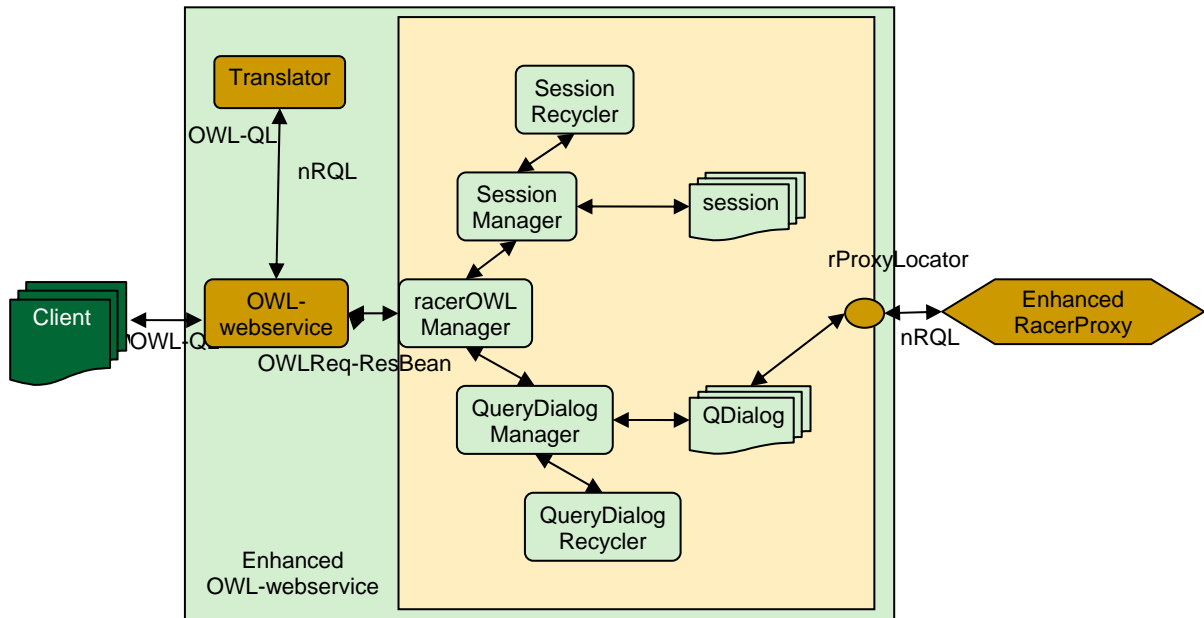


Fig APPNEDIX C. Interface between RACER Client and the RACER Proxy

This figure is taken from the presentation „ Enhancement of the Racer Proxy web services Interface for Iterative Query Answering“ by Tejas Doshi, (2005-01-04, STS, Hamburg)