LarKC

The Large Knowledge Collider
a platform for large scale integrated reasoning and Web-search

FP7 – 215535

D1.2.1

Initial Operational Framework

Gaston Tagni (VUA), Annette ten Teije (VUA)
Frank van Harmelen (VUA), Barry Bishop (UIBK)
Mick Kerrigan (UIBK), Georgina Gallizo (HLRS)
Axel Tenschert (HLRS), Luka Bradesko (CycEur),
Vassil Momtchev (OntoText), Atanas Kiryakov (OntoText)

Document Identifier:	LarKC/2008/D1.2.1
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	version 1.0.0
Date:	October 30, 2008
State:	final
Distribution:	public

Executive Summary

This document reports on the ongoing work being conducted as part of the activities in WP1 and WP5 regarding the design and implementation of the LarKC platform. It provides a description of the first version of the platform and its underlying architecture. More specifically, we define the five types of plug-ins that constitute the platform and for each of them a description of its behaviour and interfaces (I/O types) is provided. Additionally, we provide a description of the different usage phases of the platform, i.e. a description of how different users will interact with the platform. The document also introduces the datamodel adopted by the platform. To help validate the initial architecture we have studied the three uses cases considered in LarKC and identified how the different use cases can be supported by the current version of the platform. We also report on a simple LarKC pipeline that has been built to validate the design decisions from this document. This document also reports on some of the discussions that have taken place during the design of the first version of the platform. Architectural design aspects such as remote invocation, support for anytime behaviour and data storage layer are discussed. We also describe a prototype implementation of the current version of the platform.

Document Information

IST Project Number	FP7 – 215535	**Acronym**	LarKC
Full Title	Large Knowledge Collider		
Project URL	`http://www.larkc.eu/`		
Document URL			
EU Project Officer	Stefano Bertolo		

Deliverable	**Number**	1.2.1	**Title**	Initial Operational Framework
Work Package	**Number**	1	**Title**	Conceptual Framework

Date of Delivery	**Contractual**	M7	**Actual**	31-Oct-08
Status	version 1.0.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Gaston Tagni (VUA), Annette ten Teije (VUA), Frank van Harmelen (VUA), Mick Kerrigan (UIBK), Barry Bishop (UIBK), Georgina Gallizo (HLRS), Axel Tenschert (HLRS), Luka Bradesko (CycEur), Vassil Momtchev (OntoText), Atanas Kiryakov (OntoText)			
Resp. Author	Gaston Tagni (VUA)		**E-mail**	gtagni@cs.vu.nl
	Partner	VUA	**Phone**	+31 (20) 59-87753

Abstract (for dissemination)	This document reports on the ongoing work being conducted as part of the activities in WP1 and WP5 regarding the design and implementation of the LarKC platform. It provides a description of the first version of the platform and its underlying architecture. More specifically, we define the five types of plug-ins that constitute the platform and for each of them a description of its behaviour and interfaces (I/O types) is provided. Additionally, we provide a description of the different usage phases of the platform, i.e. a description of how different users will interact with the platform. The document also introduces the datamodel adopted by the platform. To help validate the initial architecture we have studied the three uses cases considered in LarKC and identified how the different use cases can be supported by the current version of the platform. We also report on a simple LarKC pipeline that has been built to validate the design decisions from this document. This document also reports on some of the discussions that have taken place during the design of the first version of the platform. Architectural design aspects such as remote invocation, support for anytime behaviour and data storage layer are discussed. We also describe a prototype implementation of the current version of the platform.
Keywords	LarKC platform, LarKC Plug-ins, Web scale reasoning, Semantic Web

Project Consortium Information

Acronym	Partner	Contact
Semantic Technology Institute Innsbruck http://www.sti-innsbruck.at	STI	Prof. Dr. Dieter Fensel Semantic Technology Institute (STI) Innsbruck, Austria E-mail: dieter.fensel@sti-innsbruck.at
AstraZeneca AB http://www.astrazeneca.com/	ASTRAZENECA	Bosse Andersson AstraZeneca Lund, Sweden E-mail: bo.h.andersson@astrazeneca.com
CEFRIEL SCRL. http://www.cefriel.it/	CEFRIEL	Emanuele Della Valle CEFRIEL SCRL. Milano, Italy E-mail: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O. http://cyceurope.com/	CYCORP	Michael Witbrock CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O., Ljubljana, Slovenia E-mail: witbrock@cyc.com
Hchstleistungsrechenzentrum, Universitaet Stuttgart http://www.hlrs.de/	HLRS	Georgina Gallizo Hchstleistungsrechenzentrum, Universitaet Stuttgart Stuttgart, Germany E-mail : gallizo@hlrs.de
Max-Planck-Institut fr Bildungsforschung http://www.mpib-berlin.mpg.de/index_js.en.htm	MAXPLANCKGESELLSCH	Dr. Hael Schooler, Max-Planck-Institut fr Bildungsforschung Berlin, Germany E-mail: schooler@mpib-berlin.mpg.de
Ontotext Lab, Sirma Group Corp. http://www.ontotext.com/	ONTO	Atanas Kiryakov, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: atanas.kiryakov@sirma.bg
SALTLUX INC. http://www.saltlux.com/EN/main.asp	Saltlux	Kono Kim SALTLUX INC Seoul, Korea E-mail: kono@saltlux.com
SIEMENS AKTIENGESELLSCHAFT http://www.siemens.de/	Siemens	Dr. Volker Tresp SIEMENS AKTIENGESELLSCHAFT Muenchen, Germany E-mail: volker.tresp@siemens.com
THE UNIVERSITY OF SHEFFIELD http://www.shef.ac.uk/	Sheffield	Prof. Dr. Hamish Cunningham THE UNIVERSITY OF SHEFFIELD Sheffield, UK E-mail: h.cunningham@dcs.shef.ac.uk
VRIJE UNIVERSITEIT AMSTERDAM http://www.vu.nl/	Amsterdam	Prof. Dr. Frank van Harmelen VRIJE UNIVERSITEIT AMSTERDAM Amsterdam, Netherlands E-mail: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTITUTE, BEIJING UNIVERSITY OF TECHNOLOGY http://www.iwici.org/	WICI	Prof. Dr. Ning Zhong THE INTERNATIONAL WIC INSTITUTE Mabeshi, Japan E-mail: zhong@maebashi-it.ac.jp

INTERNATIONAL AGENCY FOR RESEARCH ON CANCER http://www.iarc.fr/	IARC2 	Dr. Paul Brennan INTERNATIONAL AGENCY FOR RESEARCH ON CANCER Lyon, France E-mail: brennan@iarc.fr

Table of Contents

List of Abbreviations

DUNS	Data Universal Numbering System
GSI	Grid Security Infrastructure
MPI	Message Passing Interface
OASIS	Organization for the Advancement of Structured Information Standards
OGSA-DAI	Open Grid Services Architecture- Data Access Integration
QoS	Quality of Service
SAML	Security Assertion Markup Language
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
WSDL	Web Service Description Language
WSS	Web Service Security
UML	Unified Modeling Language
LLD	Linked Life Data
CPT	Clinical Project Team
ORDI	Ontology Representation and Data Integration Framework
RDF	Resource Description Framework

1 Introduction

The principal goal of LarKC is to build a platform for massive distributed incomplete reasoning at Web-scale. Such a platform will be realized through a pluggable architecture that gives users the possibility to develop plug-ins in order to test heuristics and techniques from diverse areas such as Web search, Semantic Web, databases, machine learning and cognitive science, among others. The platform will run on a computing cluster and will also consider alternative computing paradigms such as "computing-at-home".

This document reports on the ongoing work being conducted as part of the activities in WP1 and WP5 regarding the design and implementation of the LarKC platform. It provides a description of the first version of the platform and its underlying architecture. More specifically, we define the five types of plug-ins that constitute the platform and for each of them a description of its behaviour and interfaces (I/O types) is provided. Additionally, we provide a description of the different usage phases of the platform, i.e. a description of how different users will interact with the platform. To help validate the initial architecture we have studied the three uses cases considered in LarKC and identified how the different use cases can be supported by the current version of the platform. More specifically, for each use case we produced a storyboard describing the key functionality required from the LarKC platform in order to implement the use case. Additionally, for every storyboard we identified a series of plug-ins that can be used for implementing applications that automate the scenario.

Besides providing a description of the initial architecture this document also reports on some of the discussions that have taken place during the design of the first version of the platform. In particular we discuss issues concerning remote invocation of plug-ins and the support for anytime behaviour. The purpose of these discussions is to contribute to the ongoing architectural discussions taking place in WP1 and WP5 and not as final architectural decisions for the LarKC Framework.

Many issues related to the functionality and design of the platform still remain open, for example QoS aspects and the data-storage component. These and other issues will be addressed in a later version of this document due in month 33.

Structure of this document: Chapter 2 introduces the LarKC architecture by providing a description of each of the plug-in types supported by the platform. Each plug-in is described in terms of its API indicating the corresponding input and output. This chapter also includes a description of the datamodel adopted by the LarKC platform. In Chapter 3 we provide a description of the use cases considered in LarKC in terms of the plug-in types defined in Chapter 2. For every use case we analyze its functional requirements and propose a possible set of plug-ins to support the different application scenarios in each of them. Chapter 4 discusses issues concerning remote invocation in the context of the LarKC platform. We identify and analyze the remote issues affecting the LarKC framework. We provide a preliminary analysis and some initial thoughts about the potential implications of this distributed architecture. Some hints are given on current standards and specifications that could solve the identified issues. In Chapter 5 we tackle another important issue related to the LarKC platform, namely the support for anytime behaviour. In particular we discuss different programming models that could be

used by LarKC to achieve *anytime behaviour*. Chapter 7 describes a prototype implementation of the current version of the platform. Finally, Chapter 8 concludes the report.

2 The LarKC Platform

This chapter reports on the current state of the LarKC architecture. We provide the definition of the different plug-in types supported by the platform and describe them in terms of their functionality and their API, indicating their input and output. This chapter also introduces an overview of the datamodel adopted by the LarKC platform. We conclude the chapter with an example scenario that shows how the various components can be combined and used to handle user queries.

2.1 Overview of the LarKC Architecture

In order to support massive distributed incomplete reasoning at Web-scale the LarKC platform provides a plug-in based framework. These components or plug-ins can be orchestrated and executed according to the algorithmic schema depicted in Figure 2.1, which corresponds to the pipeline illustrated in Figure 2.2(a). This first organization of plug-ins however has some limitations. For example, in some situations it may be necessary to be able to return to any intermediate step in the pipeline after deciding. The pipeline configuration illustrated in Figure 2.2(a) does not allow this as the control flows back to a single plug-in after the *Decider* has been executed. In order to solve this problem a second version of the pipeline allows the control to flow from the *Decider* to any of the other plug-ins in the previous steps of the execution. Figure 2.2(b) illustrates this configuration. This second version still lacks flexibility in the sense that control decisions may need to be made in between steps rather than at the end of the pipeline. For example, in some scenarios it may be necessary to decide what to do after the *Selection* plug-in has been executed and before the *Reasoner* plug-in is executed. Therefore, the third version of the pipeline solves this problem by considering the *Decider* plug-in as a special type of component or, "meta"-component, in order to enable quality/resource/control decisions at any point in the pipeline. Figure 2.2(c) depicts the third and currently adopted version of the pipeline. In the picture, the dotted arrows are control-flow, the solid arrows are data-flow. Also, the picture leaves out the data-storage layer, to which all components need to have read/write access. Issues regarding the data-storage layer will be discussed later in this deliverable and further elaborated in Deliverable 1.2.2. A description of each plug-in type is given in section 2.2.

```
repeat
  obtain a selection of data;                    (IDENTIFY)
  transform to an appropriate representation;    (TRANSFORM)
  draw a sample;                                 (SELECT)
  reason on the sample;                          (REASON)
  if more time is available                      (DECIDE)
     and/or the result is not good enough        (DECIDE)
   then increase/decrease the data selection
   else exit
  end
```

Figure 2.1: Simple algorithmic schema of the LarKC platform

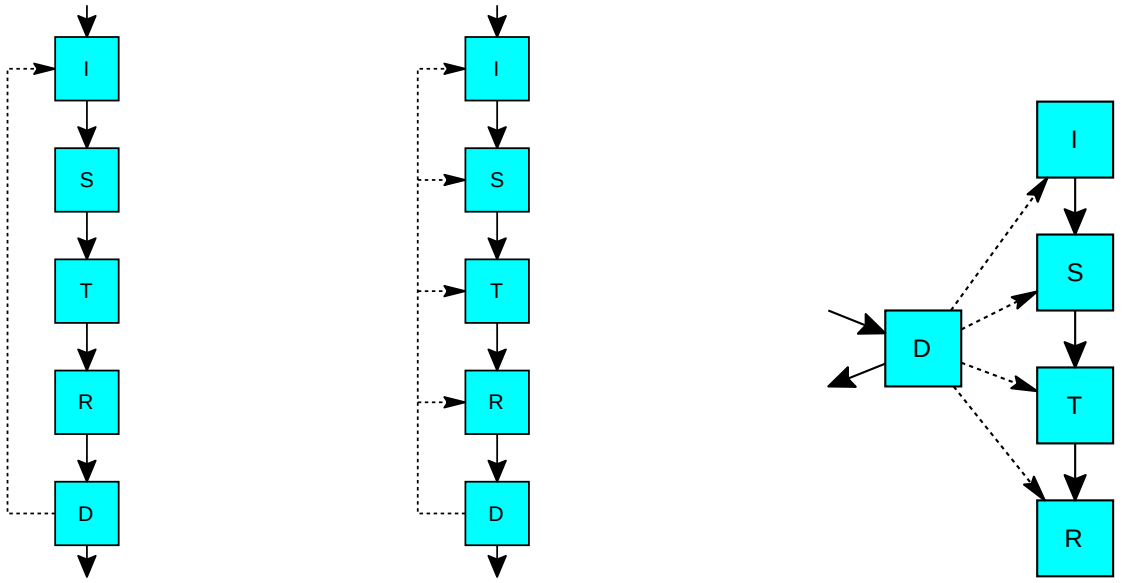

Figure 2.2: LarKC pipeline. Dotted arrows represent control-flow while solid arrows are data-flow

2.1.1 Usage of the LarKC Platform

The interaction between users and the LarKC platform will occur according to one of the following interaction patterns:

- **Plug-in Construction:** One possible way to interact with the platform is as a plug-in designer/writer. During this phase, designers will write single plug-ins and deploy them in the platform for future use.
- **Platform Configuration:** Another interaction scenario involves users (platform configuration designers) writing scripts to combine existing plug-ins in order to solve a given task. For example, users may write a Decider plug-in that combines *Selection*, *Identify*, *Transform* and *Reasoning* plug-ins to answer SPARQL queries.
- **Platform Invocation:** Finally, end users will be able to use the services provided by the platform, i.e. a particular plug-in configuration to execute SPAQRL queries. This interaction pattern also includes application designers that implement applications that need to use the services provided by the platform.

Notice that these user-platform interaction patterns could be combined in a *LarKC life cycle* involving a *construction phase* followed by a *configuration phase* and finally an *invocation phase*. Notice also that actors may not interact with the LarKC platform in all the phases, e.g. plug-in writers may not need to configure or use/invoke the platform's services at all.

In order to facilitate these interaction patterns the LarKC platform will provide support for:

- the implementation of plug-ins by providing access to data repositories, facilities for plug-in registration and discovery and by providing a framework (or set of APIs) that allows designers and programmers to implement different types of plug-ins.
- the configuration/combination of existing plug-ins in order to implement reasoning tasks capable of dealing with large scale and distributed data sources. Such combination of plug-ins will be supported by the definition of a specific type of plug-in called *Decider*. A Decider plug-in will contain the specification of control-flow dependencies among other plug-ins and will be responsible for dealing with other aspects such as remote invocation of plug-ins and anytime behavior.
- the execution of a set of plug-ins on a query and a dataset.

2.2 LarKC Plug-ins

The LarKC API consists of five major plug-in types. In this section we outline these five plug-in types, their behavior, and their interfaces. For brevity the interface of the classes are described in Java syntax; however it should be noted that this does not constrain the language in which plug-ins can be written.

All LarKC share a common super class, which is the *Plugin* class. This class provides that functionality which is common to all plug-in types. The interface of the *Plugin* class can be seen in Figure 2.3.

```
public interface Plugin {
  public String getIdentifier();
  public MetaData getMetaData();
  public QoSInformation getQoSInformation();
  public void setInputQuery(Query theQuery);
}
```

Figure 2.3: The Interface of Plugin

From the above figure it can seen that:

- All plug-ins are identified by a name, which is a string
- Plug-ins provide meta data that states the functionality that they offer
- Plug-ins provide Quality of Service (QoS) information regarding how they perform the functionality that they offer
- All plug-ins may need access to the initial query and thus a mutator is provided by specifying this query

2.2.1 Identify

Given a specific query the Identify plug-in will identify those Information Sets that could be used in order to answer the users query. The interface of the Transform plug-in can be seen in Figure 2.4. This plug-in type was originally named Retrieval (because it provides the functionality required during the retrieval step of the pipeline). However, we decided to rename it to Transform given that its main functionality is to *identify* possible data sources that can be relevant for answering a given query.

```
public interface Identifier extends Plugin{
  public Collection<InformationSet> identify(Query theQuery,
                                     Contract theContract);
}
```

Figure 2.4: The Interface of the Identify plug-in

Thus the Transform plug-in is used for narrowing the scope of a reasoning task from all available information sets to those information sets that are capable of answering the query. The plug-in takes a query as input and produces a collection of Information Sets as output according to the provided contract. The contract is used to specify the dimensions of the output, for example the number of results returned by a given call to the Transform plug-in. An example of an Identify plug-in could for example identify relevant pieces of data from a semantic data index such as Sindice [1]. Such a plug-in would take Triple Pattern Queries as input and give RDF Graphs as output. These RDF Graphs would match the input Triple Pattern query and thus would contain triples that could be used to answer the users query.

2.2.2 Transform

The Transform plug-in is capable of transforming some piece of data from one representation to another. In the LarKC API we distinguish between the transformation of queries and the transformation of Information Sets. Thus there are two transform plug-in interfaces within the API, namely the QueryTransformer and InformationSetTransfomer. The interface of each can be seen in Figures 2.5 and 2.6 respectively.

```
public interface QueryTransformer extends Plugin {
  public Set<Query> transform(Query theQuery, Contract the Contract);
}
```

Figure 2.5: The Interface of the Query Transform Plug-in

The QueryTransformer plug-in thus can take a Query as input and produce a set of queries as output according to the provided contract. The contract is used to specify the dimensions of the output, for example the number of queries to generate

[1] http://www.sindice.com

from the input query. An example of a query transformer would be a SPARQL to Triple Pattern query transformer. Such a plug-in would take a SPARQL query as input and extract triple patterns from the WHERE clause to build a triple pattern query. This plug-in was originally named abstraction plug-in and used during the abstraction phase of the pipeline. However, given that its main functionality is to *transform* data from one representation to another the name was changed to Transform plug-in.

```
public interface InformationSetTransformer extends Plugin{
  public InformationSet transform(InformationSet theInformationSet,
                                  Contract  theContract );
}
```

Figure 2.6: The Interface of the Information Set Transform Plug-in

The InformationSetTransformer plug-in thus can take an Information Set as input and provides the transformed information set as output according to the provided contract. The contract defines the dimensions of the output Information Set. An example of such a transformer would be GATE [2] or KIM [3]. Such a plug-in would take a Natural Language Document as input and would extract a number of triples from the document to create an RDF graph, which it would return as output. The contract in this case would describe how many triples the plug-in should extract from the natural language document (and potentially their types).

2.2.3 Select

The Select plug-in is responsible for narrowing the search space even further than Identify has done by taking a selection (or a sample) of the Data Set that has been made available by Identify on which reasoning should be performed. The output of a Select plug-in is a Triple Set, which is essentially a subset of the input Data Set. The interface of the Select Plug-in can be seen in Figure 2.7.

```
public interface Selecter extends Plugin{
  public TripleSet select (DataSet theDataSet, Contract theContract);
}
```

Figure 2.7: The Interface of the Select Plug-in

Thus the Select plug-in takes a Data Set as input, identifies a selection from this dataset according to its strategy and returns a Triple Set according to the contract. The contract is used to define the dimensions of the output. An example of a Select plug-in would be one that extracts a particular number of triples from each of the RDF graphs within the Data Set to build the Triple Set. The Contract in this case would define the number of triples to be present in the output Triple Set, or the number of triples to extract from the each of the RDF graphs in the Data Set.

[2] http://gate.ac.uk/
[3] http://www.ontotext.com/kim/

2.2.4 Reason

The Reason plug-in (or simply Reasoner) will execute a given SPARQL Query against a Triple Set provided by a Select plug-in. The interface of the Reason plug-in can be seen Figure 2.8.

```
public interface Reasoner extends Plugin{
  public VariableBinding sparqlSelect(SPARQLQuery theQuery,
                          TripleSet theTripleSet, Contract contract);
  public RdfGraph sparqlConstruct(SPARQLQuery theQuery, TripleSet
                                   theTripleSet, Contract contract);
  public RdfGraph sparqlDescribe(SPARQLQuery theQuery, TripleSet
                                  theTripleSet, Contract contract);
  public BooleanInformationSet sparqlAsk(SPARQLQuery theQuery,
                                         TripleSet theTripleSet,
                                         Contract contract);
}
```

Figure 2.8: The Interface of the Reason Plug-in

The Reasoner supports the four standard methods for a SPARQL endpoint, namely select, describe, construct and ask. The input to each of the reason methods are the same and includes the query to be executed, the triple Set to reason over and the contract, which defines the behavior of the reasoner. An example of a Reasoner plug-in would wrap the reasoning component provided by the Jena Framework [4]. In such a plug-in the data in the Triple Set is loaded into the reasoner and the SPARQL query is executed against this model. The output of these reasoning methods depends on the reasoning task being performed. The select method returns a Variable Binding as output where the variables correspond to those specified in the query. The construct and describe methods return RDF graphs, in the first case this graph is constructed according to the query and in the second the graph contains triples that describe the variable specified in the query. Finally ask returns a Boolean Information Set as output, which is true if the pattern in the query can be found in the Triple Set or false if not.

2.2.5 Decide

The Decide plug-in is responsible for building and maintaining the pipeline containing the other plug-in types, and managing the control flow between these plug-ins. The interface of this plug-in can be seen in Figure 2.9.

The interface of the Decide plug-in is very similar to that of the reason plug-in (as LarKC is a platform for large scale reasoning and as such can be viewed as a reasoner). The major difference is that actual data to reason over is not explicitly specified, at least not in the current version of the platform, as the Identify plug-in is responsible for finding the data within the pipeline.

[4] http://jena.sourceforge.net/

```
public interface Decider extends Plugin{
  public VariableBinding sparqlSelect(SPARQLQuery theQuery,
                           QoSParameters theQoSParameters);
  public RdfGraph sparqlConstruct(SPARQLQuery theQuery,
                                  QoSParameters  theQoSParameters);
  public RdfGraph sparqlDescribe(SPARQLQuery theQuery,
                                 QoSParameters theQoSParameters);
  public BooleanInformationSet sparqlAsk(SPARQLQuery theQuery,
                                 QoSParameters theQoSParameters);
}
```

Figure 2.9: The Interface of the Decide Plug-in

2.3 Datamodel

As can be seen in the previous section there are a number of different types of objects that are passed along the pipeline. In this section we provide an overview of these data types and an understanding of the relationships between them.

In the LarKC pipeline the Identify plug-in is responsible for finding resources that could be capable of answering the user's query. We refer to these resources as *Information Sets* and identify two major sub-types that can be returned by identify, namely *RDF Graphs* and *Natural Language Documents*. The Transform plug-in is responsible for transforming between different representations and one important type of Transform plug-in can extract RDF Graphs from Natural Language Documents. Thus when we reach the Select plug-in in the LarKC pipeline we are dealing with RDF Graphs that have been identified as potentially able to answer the user's query and which may have been extracted from Natural Language Documents. This collection of RDF Graphs is termed a *Dataset* and consists of a *Default Graph* (an unnamed RDF graph) along with any number of named RDF graphs. The Select plug-in is responsible for identifying a selection (or sample) from within a Dataset that is to be reasoned over to answer the user's query. This selection is termed a *Triple Set* and consists of triples from any of the RDF Graphs in the Dataset. The relationship between RDF Graphs, Datasets and Triple Sets is shown in Figure 2.10

Having selected a Triple Set, the Reason plug-in will reason over it in order to fulfill the user's query, which can be a SPARQL Select, Construct, Describe or Ask query. The return types of each of these types of queries are also Information Sets. The SPARQL Select query will return a *Variable Binding* that provides an assignment of values for each of the variables contained within the user's SPARQL query. The SPARQL Construct and Describe queries both return an RDF Graph as output. For SPARQL Construct this graph is a new graph constructed based on the user's query and for SPARQL Describe this graph is made up of triples that describe the object specified in the SPARQL query. Finally a SPARQL Ask query will return a *Boolean Information Set*, which will contain true if the pattern described in the user's query is found in the Triple Set being reasoned over or will contain false otherwise.

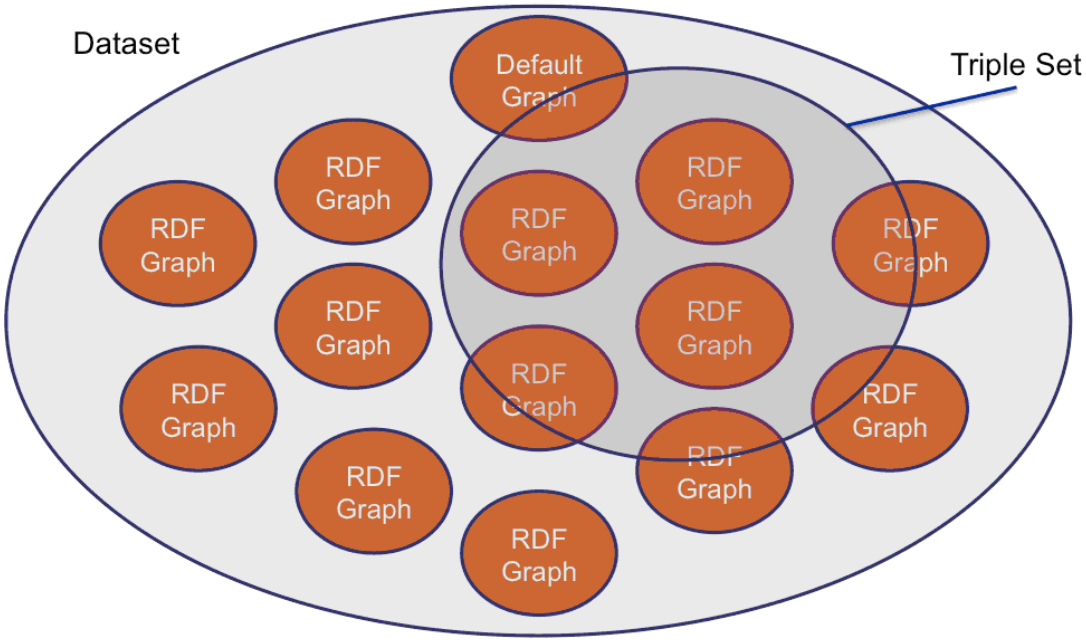

Figure 2.10: Elements of the LarKC Data Model

2.4 Example Scenario

The motivation for this storyboard comes from an attempt to try and understand what activities take place when a LarKC platform evaluates a query. It is not intended to try and identify which components will carry out particular tasks, nor even to identify what components will exist at all. Rather, the aim is to show a sequence of activities.

Consider the SPARQL query shown in Figure 2.11 which expresses the sentence "Who knows Tim Berners-Lee?":

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT ?name WHERE {
       ?person rdf:type foaf:Person .
       ?person foaf:name "Tim Berners-Lee" .
       ?person2 rdf:type foaf:Person .
       ?person2 foaf:knows ?person .
       ?person2 foaf:name ?name .
}
```

Figure 2.11: An example of a SPARQL query

In order to evaluate this query, the following steps must be completed:

1. The user passes the query to the LarKC platform.
2. A pipeline must be constructed from available components that pass the correct data-types between them.

3. The query is examined to extract keywords - any identifier from the triple patterns of the WHERE clause.

4. The keywords are passed to an Identify plug-in (a wrapper for `www.sindice.com` in this case).

5. The identify step returns URLs of RDF documents (or documents containing embedded RDF statements).

6. No data transformation is imagined for processing this query.

7. The URLs are passed to a select component that needs access to the triples, therefore it is here that the URLs must be 'dereferenced' and RDF statements downloaded, and the selection component applies its selection strategy (in this simple case returning all triples).

8. The triples are passed on to the reasoning component that executes the original SPARQL query against this data set.

9. The results of the query are returned to the user.

The storyboard described above has been implemented in the first demo of the platform, which can be tested at `http://www.larkc.eu/baby-larkc`.

3 Use-case Scenarios

In this chapter, we take the description and requirement specification of the each of use cases considered in LarKC (reported in deliverables D7b.1.1a, D7b.1.1b and D6.1) and present one possible way of combining different LarKC plug-ins in order to support the implementation of each of the scenarios. For the sake of simplicity we do not provide a detailed description of each use case. The interested reader is referred to D6.1 (*Urban Computing* use case), D7b.1.1a (*Early Clinical Development* use case) and D7b.1.1b (*Carcinogenesis Research* use case).

3.1 Use Case WP6: Urban Computing

3.1.1 Getting to Milano

This story-board use-case, used as an example study LarKC platform configuration, expects LarKC to be used by a travel planning system, which will have access to, and reason over, all of the city of Milan's traffic, event and weather data. In this particular use-case, users do not query the platform directly, but use some advisory system which calls LarKC as its reasoning engine and/or data retrieval service.

The use of the Platform in this case is divided into two modes:

- Planning mode, which starts when the user asks for possible routes to a desired destination, and
- Informing mode, which comes into action when additional data that affects previously planned trips becomes available.

Planning mode: The purpose of this mode is to find one or more possible near-optimal ways to travel from the user's current location to a destination while obeying a set of constraints. Data is retrieved from a variety of sources, all of which may return different types of results. This suggests a LarKC configuration requirement of one data transformer plug-in for each data source. Also, because this travel planning problem has particular features, for example by virtue of being, at base, a planning problem, an extended (non-default) implementation of a decider plug-in may be needed.

Table 3.1 shows the functionality expected from the LarKC platform in order to support planning. Figure 3.1 illustrates the corresponding UML diagram, describing the interactions among plug-ins.

Informing mode: The purpose of this mode is to monitor traffic status, and traffic-affecting events, and then find the planned routes that this new data affects. This task is easier in some respects than that of planning mode since only two data sources (possible identifiers) are needed: one accessing data about current planned routes, and one the traffic status data. The associated reasoning, though, is potentially quite complex, depending on the nature of the status data.

In the simplest implementation, this "Informing mode reasoning" could amount to simply returning to planning mode with a new route query, to check whether there is more appropriate route under the new traffic/weather conditions. More sophisticated plan transformation implementations are also possible.

Support for	Possible Plug-ins
Retrieve route planning data from several sources (e.g. Google maps, railroad companies, bus operators, etc.)	Identifier
Retrieve weather information	Identifier
Retrieve parking information	Identifier
Retrieve traffic information	Identifier
Integration of data	Decider
Pick the few best ways to reach the destination	Reasoner

Table 3.1: Possible Plug-ins supporting the implementation of the planning scenario

Communication between plug-ins for simple "informing" is similar to that in planning mode (Figure 3.1), except that two different queries are issued. Table 3.2 shows the functionality that is required from the LarKC platform in order to complete this reasoning task:

Support for	Possible Plug-ins
Retrieve planned routes that are still valid	Identifier
Retrieve traffic/weather conditions and status	Identifier
Select appropriate subset, i.e. Match data by locations	Selecter
Decide which routes are affected	Reasoner

Table 3.2: Possible Plug-ins supporting the implementation of the planning scenario

3.2 Use Case WP7a: Early Clinical Drug Development

The Early Clinical Drug Development use case is divided into four scenarios or subcases. In the following we describe the requirements that each of these scenarios impose on the LarKC platform and provide a possible configuration of plug-ins in order to support the implementation of applications in each of the scenarios.

3.2.1 Improve Knowledge About Disease and Patients

This scenario is divided in two operation modes: off-line and on-line or interactive mode.

Off-line mode: The purpose is to find and process new additional information from text documents and ontologies. This constitutes the base process to keep the repository updated. Table 3.3 describes the functionality epidemiologists expect/require from an application supporting this scenario. Additionally, the table shows a set of possible LarKC plug-ins that can be used for implementing the required functionality. Figure 3.2 depicts the corresponding UML sequence diagram describing a possible interaction among LarKC plug-ins in order to implement the scenario.

On-line mode: This scenario is concerned with finding and processing new additional information, exploring hypothesis through the formulation of queries

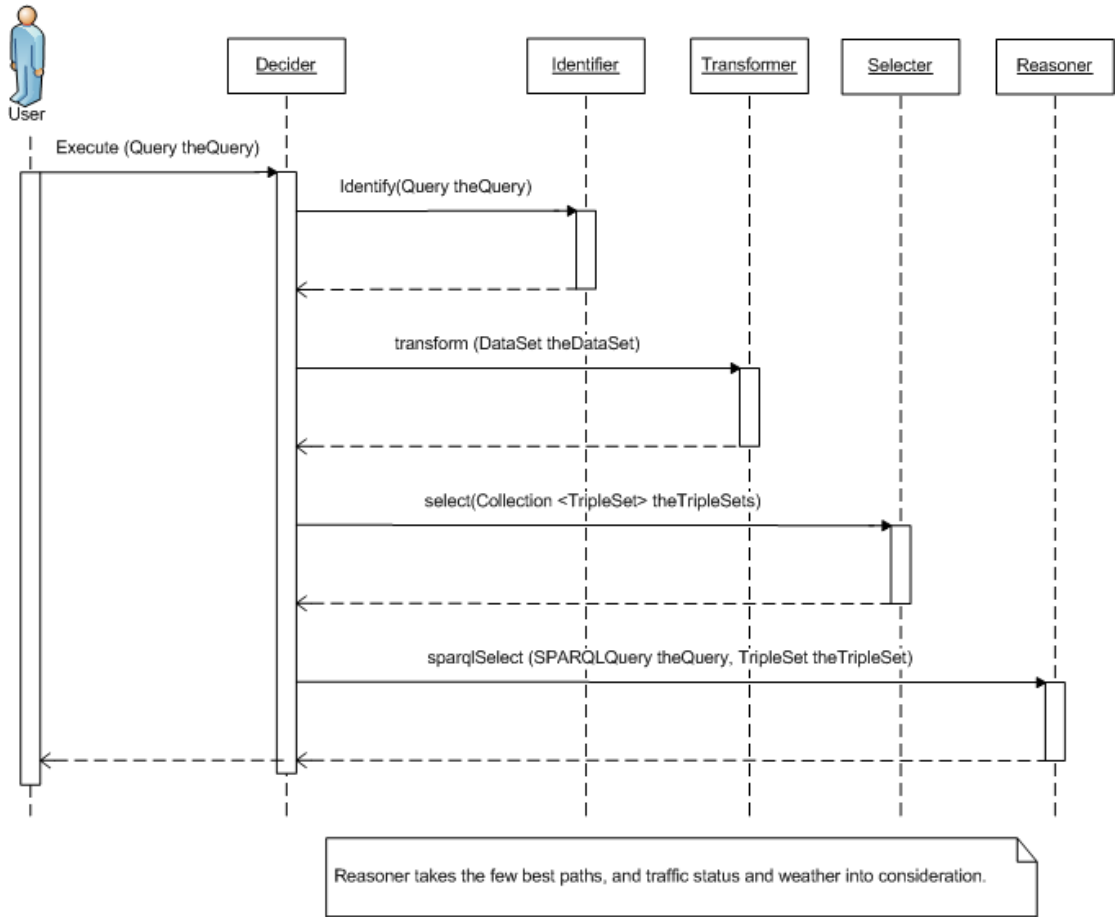

Figure 3.1: UML sequence diagram, describing the interaction of LarKC plug-ins

written in some high-level query language and enhancing the KB from the findings. Table 3.4 describes the functionality expected from the LarKC platform in order to support the implementation of this scenario. For each function a set of plug-in types supporting its implementation is proposed. Figure 3.2 depicts the corresponding UML sequence diagram describing a possible interaction among LarKC plug-ins. Note that although the same interaction pattern among plug-ins can be used for implementing this scenario, the specific plug-ins used in each case may be different. For example, to support the on-line mode described in Table 3.4 the Transform plug-in used in the pipeline needs to be able to perform name-entity recognition, which was not required in the off-line case.

This scenario is also concerned with updates of knowledge bases and data repositories. At the moment, the first version of the LarKC platform assumes update operations are implemented by the user application that is using the LarKC platform.

3.2.2 Clinical Project Team Working on a New Target

The following is a description of the functionality CPT's members expect the LarKC platform will provide to them. Along with each functionality a possible LarKC plug-in realizing it is defined. Table 3.5 describes the expected functionality from the LarKC platform along with a set of possible plug-ins supporting the

Support for	Possible Plug-ins
Retrieve (semi)structured data from different sources	Identifier
Transformation of the retrieved data into RDF	Transformer
Integration of data sources	Transformer
URI mappings	Reasoner or Decider
Select appropriate subsets of KB to work with	Selecter
Reduce redundancy with relation interpretation	Reasoner
Navigate structured information exploring relations and data sources	Decider

Table 3.3: Possible Plug-ins supporting the implementation of the off-line scenario

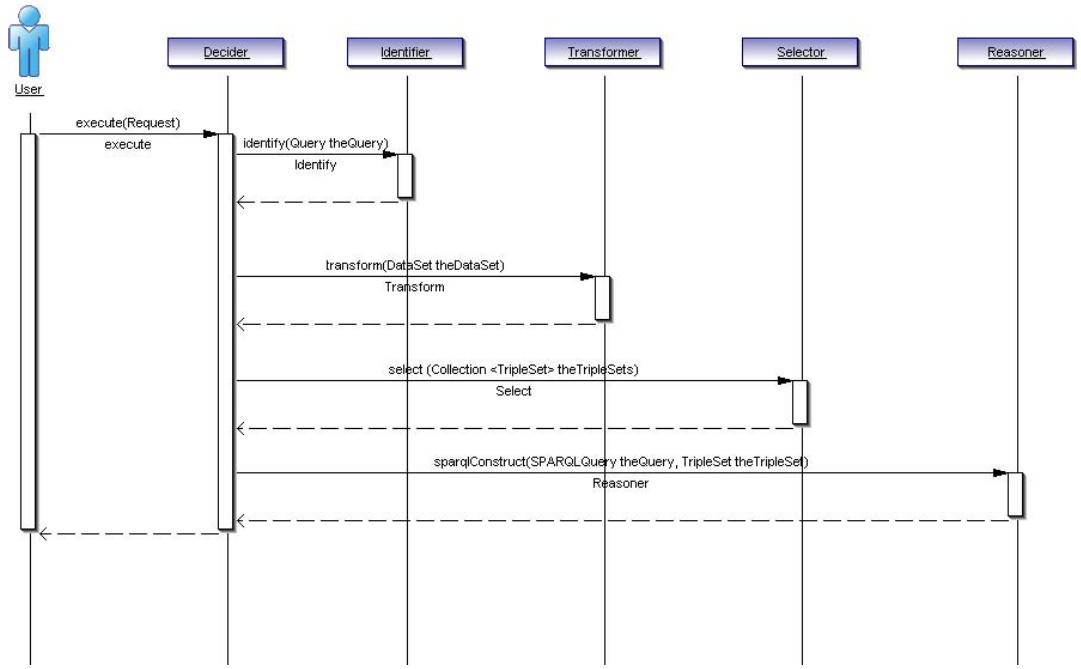

Figure 3.2: UML sequence diagram describing a possible interaction of plug-ins to implement the use case

implementation of the scenario. The same plug-in interaction pattern depicted in Figure 3.2 can be used for implementing this scenario. Once again, the specific plug-ins used will vary depending on the requirements stated in Table 3.5.

3.2.3 Identifying Biomarkers and Target Mechanisms

Table 3.6 describes the functionality users (from the Bioscientist's point of view) expect the LarKC platform will provide to them in order to be able to implement this scenario. Along with each functionality a possible set of LarKC plug-ins supporting the scenario is proposed. The same plug-in interaction pattern depicted in Figure 3.2 can be used for implementing this scenario. However, notice that the specific plug-ins used will vary depending on the requirements stated in Table 3.6.

Support for	Possible Plug-ins
Retrieve unstructured document	Identifier
Transformation of the retrieved data into RDF	Transformer
Apply name-entity recognition	Decider or Transformer
Apply semantic annotations to the ontology instances and relation extraction	Decider
Select appropriate subsets of KB to work with	Selecter
Consistency checking of extracted relations against knowledge from structured data sources	Decider
Specify queries to be evaluated against the (un)structured data, and explore different entities and their co-occurrence	Decider

Table 3.4: Possible Plug-ins supporting the implementation of the on-line scenario

Support for	Possible Plug-ins
Reuse knowledge generated by the previous scenario and extend it with results from Epidemiologist's reports	User application
Identify the mechanisms of already conducted clinical trials	Identifier
Filter available sources	Selecter
Use facet-based search interface supporting different search criteria and covering both public and in-house data sources (by applying resource classification algorithms)	Decider, Reasoner
Processing of queries written in natural language	Reasoner, Transformer

Table 3.5: Possible Plug-ins supporting the implementation of the scenario 'Clinical Project Team Working on a New Target'

3.2.4 Signal Evaluation of Adverse Drug Event Reports

Table 3.7 shows a description of the functionality users (from the Safety Expert's point of view) expect the LarKC platform will provide to them. Along with each functionality a possible LarKC plug-in realizing it is defined. The same plug-in interaction pattern depicted in Figure 3.2 can be used for implementing this scenario. However, the specific plug-ins used will vary depending on the requirements stated in Table 3.7.

3.3 Use Case WP7b: Carcinogenesis Research

The Carcinogenesis Research use case is divided in two scenarios. The first one is concerned with assisting scientists in the production of reference works (Monograph Production scenario) while the second one is concerned with assisting epidemiologists in the analysis of gene-disease association study data (the Genome Wide Association Studies scenario).

Support for	Possible Plug-ins
Import data from laboratory results	Identifier
Transform the imported data to RDF	Transformer
Identify proteins and map them to public data sets	Identifier, Decider
Filter/remove available sources	Selecter
Interpret the semantics of the relations and annotate results	User application
Test hypothesis	User application

Table 3.6: Possible Plug-ins supporting the implementation of the scenario 'Identifying Biomarkers and Target Mechanisms'

Support for	Possible Plug-ins
Collect all relevant resources	Identifier
Apply name-entity recognition	Decider, Transformer
Filter available sources, select most relevant one	Selecter
Infer relation of different adverse events	Reasoner
Interpret the semantics of the relations and annotate results	User application
User-friendly interface to explore the full information sources generated in previous sub cases	User application

Table 3.7: Possible Plug-ins supporting the implementation of the scenario 'Signal Evaluation of Adverse Drug Event Reports'

3.3.1 Monograph Production

This scenario is divided in two operation modes: off-line and user-interaction mode.

Off-line mode: The purpose of the off-line mode is to assist users in finding and processing the data (both texts and ontologies) required to answer user queries. Table 3.8 provides a summary of the tasks comprising this scenario. In the table, LLD refers to *Linked Life Data*, a platform to enable semantic data integration in early clinical and drug development process. Each of these tasks in turn imposes requirements to the LarKC platform in terms of the expected functionality. The tasks carried out in the off-line mode are the same as those carried out in the off-line mode of the Genome Wide Association Studies scenario. Table 3.8 also shows a set of possible LarKC plug-ins that can be used for supporting the scenario. The same plug-in interaction pattern depicted in Figure 3.2 can be used for implementing this scenario. However, the specific plug-ins used will vary depending on the requirements stated in Table 3.8.

Support for	Possible Plug-ins
Retrieve data for LLD	Identifier, Selecter
Transform to appropriate representations	Transformer
Integrate related LLD sources	Reasoner
Retrieve texts	Identify, Selecter
Annotate texts using LLD	User application

Table 3.8: Possible Plug-ins supporting the implementation of the off-line scenario

User-interaction mode: In the user-interaction mode users interact with an application by formulating queries using some high-level query language. The application uses the services provided by the LarKC platform to answer the queries and present the results to the end user. Table 3.9 provides a summary of the tasks comprising this scenario. The table also shows a set of possible LarKC plug-ins that can be used for supporting the scenario.

Support for	**Possible Plug-ins**
Retrieve triples matching the query	Reasoner
Present associated information: author networks, conceptual links etc.	User application, Reasoner
Navigate to related triples and texts	User application, Reasoner

Table 3.9: Possible Plug-ins supporting the implementation of the user-interaction scenario

3.3.2 Genome Wide Association Studies

As with the previous scenario, this scenario is divided in two operation modes: *off-line mode*, which is handled as in the previous scenario and, a *scripted* mode. The off-line mode is similar to the off-line mode in the monograph production scenario described before.

Scripted mode: Table 3.10 provides a summary of the tasks comprising this scenario as well as a set of possible LarKC plug-ins that can be used for supporting the scenario.

Support for	**Possible Plug-ins**
Query generation and expansion	Decider
Knowledge base querying	Decider
Paper retrieval and search	Identify, Select

Table 3.10: Possible Plug-ins supporting the implementation of the scripted mode

4 Issues Concerning Remote Invocation

One of the options for the deployment of LarKC is to consider a distributed architecture, where platform and different plug-ins are located at remote locations from each other. This distributed architecture has some implications affecting the design of the different components, such as trust and security, data transfer, etc. In the following sections we identify and analyze the remote issues affecting the LarKC framework. This is a preliminary analysis and some initial thoughts about the potential implications of this distributed architecture. Some hints are given on current standards and specifications that could solve the identified issues. Currently an architectural discussion is ongoing as part of WP1 and WP5 and therefore, the suggestions in this document must be taken as so, and not as final architectural decisions for the LarKC Framework.

4.1 Trust and Security

The aim of this section is to identify the trust and security related issues that the LarKC framework needs to address when it is deployed following a distributed architecture. In the following subsections, different phases of the LarKC framework deployment and execution are analyzed from the trust and security viewpoint. There are different standards to manage the security in a distributed environment. One of the standardization groups dealing with security is the WSS (Web Services Security) Technical Committee in OASIS [1], which includes specifications for different security related aspects, such as SOAP Message Securtiy, X.509 certificates, SAML Token Profile among other specifications.

Registration of external plug-ins

In the case where the LarKC plug-ins are located in remote sites with respect to the LarKC platform, there must be a process of registration or announcement that the plug-in is available and under which conditions. One of the essential requirements is the compatibility between the a plug-in interface and the platform interface. But besides interface compatibility, it must be ensured that the remote plug-in is a trusted entity of the platform. Therefore, it becomes necessary to setup a trust and security framework in order to ensure the authentication of external plug-ins and authorization by the LarKC platform.

Execution of a plug-in in a distributed environment

In the cases when the plug-in algorithm can be parallelized, one of the possibilities to increase the performance of its execution is to run it over distributed resources, in a thinking@home fashion. In this case, it must be ensured that the distributed resources are trusted entities to the platform, or to the central entity that manage the execution of plug-ins. The distributed execution of plug-ins is a subject that raises questions in several aspects such as confidentiality of data and confidentiality of connections between distributed resources. These issues will be

[1] `http://www.oasis-open.org/committees/wss`

further analyzed in future LarKC deliverables. Ideas from similar projects can be taken into account. For example the BOINC project [2] may provide some interesting ideas regarding security issues.

Execution of plug-ins in a cluster environment

In the case when a certain plug-in is designed to be executed within a cluster environment (for efficiency or other reasons, in the case of parallelizable algorithms) trust and security issues must be considered for the communication between the cluster and the LarKC components located outside the cluster (the LarKC platform, other LarKC plug-ins, external data storage). The way we think to use the cluster environment is basically that a plug-in (we will call cluster-plug-in) is deployed there and is executed from outside, triggered by the LarKC platform (control flow). Data (or references to that data) is transmitted from external plug-ins or from the platform to the cluster-plug-in and viceversa. Therefore we must consider security issues in all these phases of the cluster-plug-ins life cycle:

- Deployment of the plug-in inside the cluster.
- Data (or reference to data) flow between the cluster plug-in and external plug-ins or data storage components, data (or reference to data) flow between the cluster plug-in and the LarKC platform (considering bi-directional flow in both cases).
- Control-flow between the cluster plug-in and the LarKC platform (bi-directional).

The external components must have the necessary rights to access the cluster and must be trustable to be accessed by the cluster. To ensure a secure communication, the connection to the cluster is restricted by a firewall which requires a login of the user (in our case, an external plug-in or the LarKC platform). An identification of the external user is mandatory. In the same way, every component outside the cluster that need to be accessed by the cluster-plug-in has to be a trusted entity for the cluster. Therefore, all components interacting with the cluster must be authorized against it. Figure 4.1 depicts this scenario.

A cluster (inside the square) consists of the worker nodes (compute nodes) plus a head node. Login from the outer world is only possible through the head node, never to the worker nodes. In some computing centers the queuing system is also run on the head node. In the picture above, the queuing system is located in a separate machine, behind another firewall.

Another additional possibility to ensure the required security from the cluster viewpoint is to build temporary storage nodes inside the barriers of the cluster. If such storage nodes are inside the Cluster security issues such as authentication and registration for external users through the firewall are not necessary. When the process is finalized the stored data are removed from the Cluster. In general we determine that it is mandatory to have user authentication/authorization (e.g. registration, password, trusted IP) which permits a human user, a plug-in or the

[2] `http://boinc.berkeley.edu/`

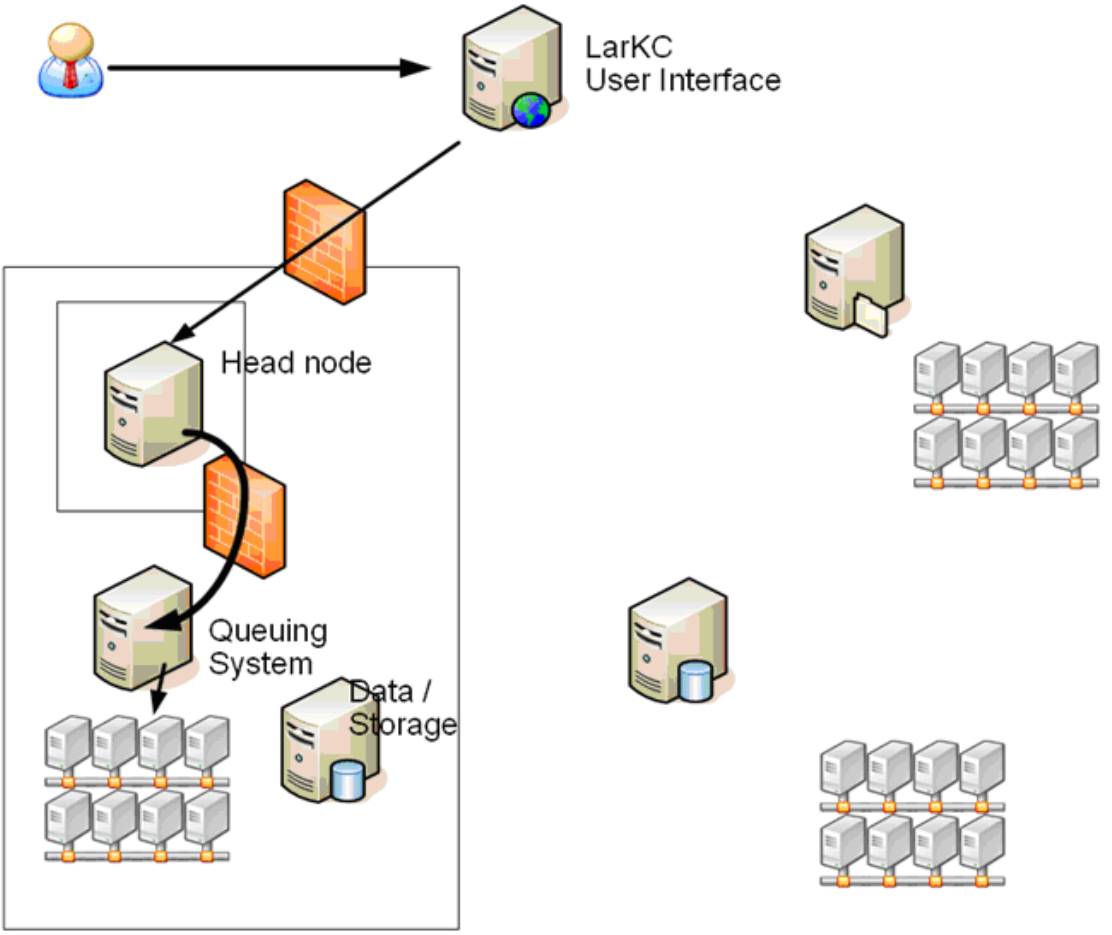

Figure 4.1: Access to a Cluster environment

LarKC platform itself to access the cluster and which allows a secure data or control flow transfer.

Secure communication between distributed plug-ins

In this section we must consider which flows are transmitted between plug-ins and how. According to the latest discussions within WP5, we can distinguish between two different kind of flows: data and control. All control flows must go through the Decider plug-in and only data (or references to data) flows are transmitted directly between the rest of plug-ins.

When two plug-ins communicate with each other it has to be ensured that both parties are trustable. In this sense, it has to be defined how authentication is performed between them. The proposed solution is that the authentication task must be done in the registration phase, that is, when the plug-in is registered in the LarKC platform. We will then assume that all plug-ins registered within a certain instance of the LarKC platform have been already authorized to belong to the LarKC framework and are therefore a trusted party. This means that all parties which are involved in the communication process are who they claim to be. Furthermore, besides authentication, authorization has to be ensured as well. For the authorization a mechanism is needed that decides when a user is authorized to perform a certain task and when not.

Authorization and authentication are closely related to each other because if a certain party is authorized for a certain task it is essential that the party is the one he claims to be he is. Security in the data transmitted must be also ensured. We must determine then in which parts of the transmission path the security may be affected (e.g. when the communication is performed through public links, like the Internet). In this cases, the level of confidentiality or privacy required for the concrete data will be the key to determine the level of security to be applied to the communication path (solutions such as data encryption can be adopted). Besides confidentiality, integrity of the data must be also ensured. This means that the receiving end must be able to know for sure that the received message is exactly the one that the transmitting end has sent to him.

Furthermore, the privacy of a communication has to be considered. A communication has to be private in the way that only the sender and the receiver should be able to understand the conversation. Through privacy a third party which might eavesdrop the conversation is not able to make any sense out of the communication massage.

4.2 Plug-in registration and discovery

The LarKC platform must allow the registration of new plug-ins located at remote sites. In this case there must be a process of registration or announcement that the plug-in is available and under which conditions. Furthermore, to be more flexible the platform should support registration of plug-ins at runtime. How this can be achieved in the LarKC platform is still an open issue that will be analyzed in more details in future deliverables.

One of the essential requirements for the registration of plug-ins is to be able to describe not only the functional aspect of plug-ins but also their non-functional aspect. This description will also allow the platform to discover plug-ins based on different properties. Such description of plug-ins will be expressed expressed in what we call "the plug-in description language". Within the plug-in description there must be information such as type of plug-in, functional parameters, QoS parameters, information regarding contract and context of the execution. The concrete definition of the plug-in description is a work in progress within WP1 and WP5 and it will be part of future deliverables of the LarKC project.

As mentioned before, once the plug-ins register in the platform (or express in some way their availability to the platform), a mechanism to find the available plug-ins must be defined. The location of resources in large scale heterogeneous networks is a complex task. To handle this, some kind of Service Discovery mechanism is needed.

One widely accepted service discovery mechanism among the web services community is UDDI [3]. UDDI is closely related to SOA and it is used as a standardized registry service for Web Services which is sponsored by OASIS. UDDI ensures the publication of service listing and discovers each other service listings and defines in which way the services interact over the Internet. It is an open industry initiative

[3] `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`

and it contains SOAP-interfaces.There are three approaches available in UDDI how the service discovery takes place [1] [4]:

1. **White Pages Approach:** The White Pages Approach includes basic information similar to a phone book. Information about the identity of the Service Provider are supported such as business area, contact details, contact person and a unique company classification number which is a D-U-N-S number.

2. **Yellow Pages Approach:** The Yellow Pages Approach orders the Web Services for a specific purpose. This approach is related to the idea of a business directory.

3. **Green Pages Approach:** The Green Pages Approach provides technical information about the services which are exposed by the business. The descriptions of the interfaces of the Web Services are used to handle the services. This seems to be the most appropriate approach to handle LarKC plug-ins.

When we think of the LarKC plug-ins modeled as Web services, UDDI might be a useful solution for managing the discovery of plug-ins by the Decider plug-in ("meta-reasoner" of the platform). It is possible to store the services (in this case the LarKC plug-ins) in a UDDI Registry. In the case of LarKC the Decider plug-in will be able to find the required plug-in by contacting the UDDI Registry.

4.3 Heterogeneity

As part of the LarKC concept, external plug-in developers should be able to plug their own plug-ins and integrate them together with other developers' plug-ins. Therefore, it would be interesting to ensure that the plug-ins and platform interfaces are independent of implementation language and execution platform, in order to ensure the compatibility of heterogeneous components, implemented in different programming languages and running on different platforms.

The Web services approach to model the plug-ins seems to be and optimal and very convenient solution to cope with this issue. The use of a description language such as WSDL to describe the plug-ins as Web services allows the integration and interoperability of heterogeneous services, making the LarKC architecture independent from any specific implementation language or running platform of that service. However, the Web service technology stack has its disadvantages too and in particular, it may not be well suited for the high-speed, high-volume message passing that might be needed in some of the LarKC use-cases. Therefore, current discussions are focused on whether it is necessary to go beyond WSDL for the description of plug-ins and add some domain specific language to describe plug-ins and their properties. Future deliverables in the LarKC project will consider this issue in more details.

[4] `http://www.irmosproject.eu/Deliverables/Default.aspx`

4.4 Data Transfer between remote components

At the time of writing this report, two different types of flows are being considered between remote components:

- **Data flow:** directly between plug-ins.
- **Control flow:** between the Decider plug-ins and the rest of plug-ins.

It will be avoided, as far as possible, to transmit big amounts of data between plug-ins, for which we are applying the policy of "lazy de-referencing", consisting on passing references to the data sources as long as it is possible.

If we consider a distributed web services architecture, a possible solution to manage data is OGSA-DAI [5]. OGSA-DAI is a middleware product that allows data resources, such as relational or XML databases, to be accessed via web services.

Some of the the main motivations for the development of OGSA-DAI are:

- To allow different types of data resources such as relational databases, XML repositories and files to be accessed from Grids.
- To provide a way of querying, updating, transforming and delivering data via Web services.
- To provide consistent and data-independent access to data sources as well as metadata.
- To support data integration from various data resources.
- To enable composition of Web services to provide higher-level Web services that support data federation and distributed query processing.

A protocol for data transfer, compatible with the OGSA-DAI approach is GridFTP. It is used in the area of Distributed Computing. It is an extension of the FTP (File Transfer Protocol) standard which ensures the use of FTP in a Distributed environment. It is supported by the Open Grid Forum (OGF) [6].

The aim of GridFTP is to guarantee a speedy data transfer of very large amounts of data. Regarding to a Grid/Distributed environment the transfer of large data amounts is, of course, often a necessity. Therefore, GridFTP is an adequate solution for data transfer, especially when thinking about reliability and executing time of transfer. Furthermore, GridFTP bridges the gap between data storage and accessing a system by defining a protocol for data transfer. Beyond that GridFTP provides security aspects through in-built security. GridFTP enhances FTP by aspects such as security with GSI, third party transfers, parallel and striped transfer, partial file transfer, fault tolerance, automatic restart of transfers and automatic TCP optimization.

[5] http://www.ogsadai.org.uk/about/ogsa-dai/
[6] http://www.ogf.org/

4.5 Synchronization and communication between parallel tasks

Different parallelization approaches are being considered for LarKC:

- Parallelization between plug-ins.
- Parallelization inside the plug-ins.

In both cases, we have to consider a way to communicate and synchronize between the parallel tasks that may be dependent on each other.

There may be a task which might depend on other tasks processed at same time. MPI is a de facto standard for the exchange of messages for parallel processing on distributed systems which is supported by the MPI-Forum [7].The goals of MPI are high performance, scalability and portability. In general an MPI application consists of several processes which are communicating with each other. All of these processes are started in parallel at the same time. The processes are working together on one task and they are exchanging messages with each other. The advantage of MPI is that the message exchange can take place among several computers.

[7] http://www.mpi-forum.org/

5 Support for Anytime Behaviour

In this section we discuss a number of programming models that could be used by LarKC to achieve *anytime behaviour*. As such, it will first try to define anytime behaviour and then present a set of potential design solutions (programming models). Each solution will have its own advantages and disadvantages. Through this discussion, it is hoped that the consortium will choose the approach most suitable for LarKC and this model will become the de facto standard that all writers of software components must adhere to.

In essence, LarKC hopes to achieve fast search/reasoning by using a combination of:

- Approximate techniques.
- Novel techniques (borrowed from other disciplines), and
- Parallel execution.

Parallel execution involves the simultaneous execution of more than one execution context, or thread. For a single (hardware) processor system, this can sometimes allow for a simpler and more efficient design. However, it is on multi-processor systems that software designed for parallel execution can expect to achieve its scalability goals, i.e. more processors implies faster execution or more data processed. In the context of LarKC and a multi-processor (and therefore a multi-threaded) environment, it naturally follows that a desired behaviour of a LarKC platform would be that the 'user' could terminate a search/reasoning task at anytime and still have some meaningful answer (or set of results). Such a behaviour is called *anytime*. The principal advantage, is that it allows the user to decide how best to trade response times with accuracy/quantity of results. Further, the LarKC platform is constructed from many components that work together (under the guidance of a Decide component) to deliver results to the user. These components will in turn need to communicate and will likely also pass 'intermediate results' between themselves.

In order to support anytime behaviour among plug-ins and also between the platform and the user, the API introduced in section 2.2 are defined in such a way that the communication of streams of data among components is possible. More concretely, for every plug-in type that naturally produces something of type X, the interface defines that the plug-in actually produces a stream of X's. The maximum size of the stream is given by the contract parameter. For example, the Select plug-in has as contract parameter the number of triples to be returned. When called, it starts producing a stream of triples up to the size of the contract parameter. This combination of contract parameter and streaming output gives the consumer a good control over how many data are being maximally produced, in a single call. The consumer can choose between many calls with a small contract parameter (each producing short streams), or a single call with a large contract parameter, producing a single large stream. Of course, a particular design must be chosen to implement such non-blocking streaming output.

In the following sections we discuss different design alternatives aimed at supporting anytime behaviour in LarKC. We first consider the case of anytime behaviour that exists at the boundary of LarKC, i.e. between the user and platform.

It may well be that one solution fits all, so that we can use a single anytime model throughout LarKC as a whole. However, after considering the user-LarKC interface, the section will be further expanded with an analysis of the communication requirements between LarKC components.

5.1 Interface between LarKC and User

The interface between the LarKC platform and the users should be defined with the following design goals in mind:

- The LarKC interface should be as simple as possible to give the user maximum flexibility to integrate with LarKC.
- The user should not be forced to use unacceptably complex synchronisation mechanisms (preferably none at all).
- The user thread should not block indefinitely when making a request to a LarKC platform.

5.1.1 Callback

In some ways, this is the simplest model of all. The basic idea is similar to the one implemented by the Observer design pattern, i.e. to get notifications of events, one must create a class that supports a particular Observer interface and pass an instance of this class to the "thing" doing the notification. Figure 5.1 shows the UML class diagram that models a callback scenario.

The sequence of events in the LarKC case would be as follows (the corresponding UML sequence diagram is depicted in Figure 5.2):

1. User creates an object to receive asynchronous results (a receiver).
2. User passes this receiver object with a request (SPARQL query) to LarKC (and returns immediately).
3. LarKC does some processing and starts to return results by calling methods on the receiver object (LarKC thread, not user thread).
4. The receiving object does whatever it needs with the results.

Advantages of the Callback-based Method

- Very simple model.
- Well understood and similar to existing java paradigms (although swing is single-threaded).
- User thread is at no time blocked in LarKC components (either waiting for a LarKC response or polling an object for a response).
- User has complete control over synchronisation mechanism used.

Disadvantages of the Callback-based Method

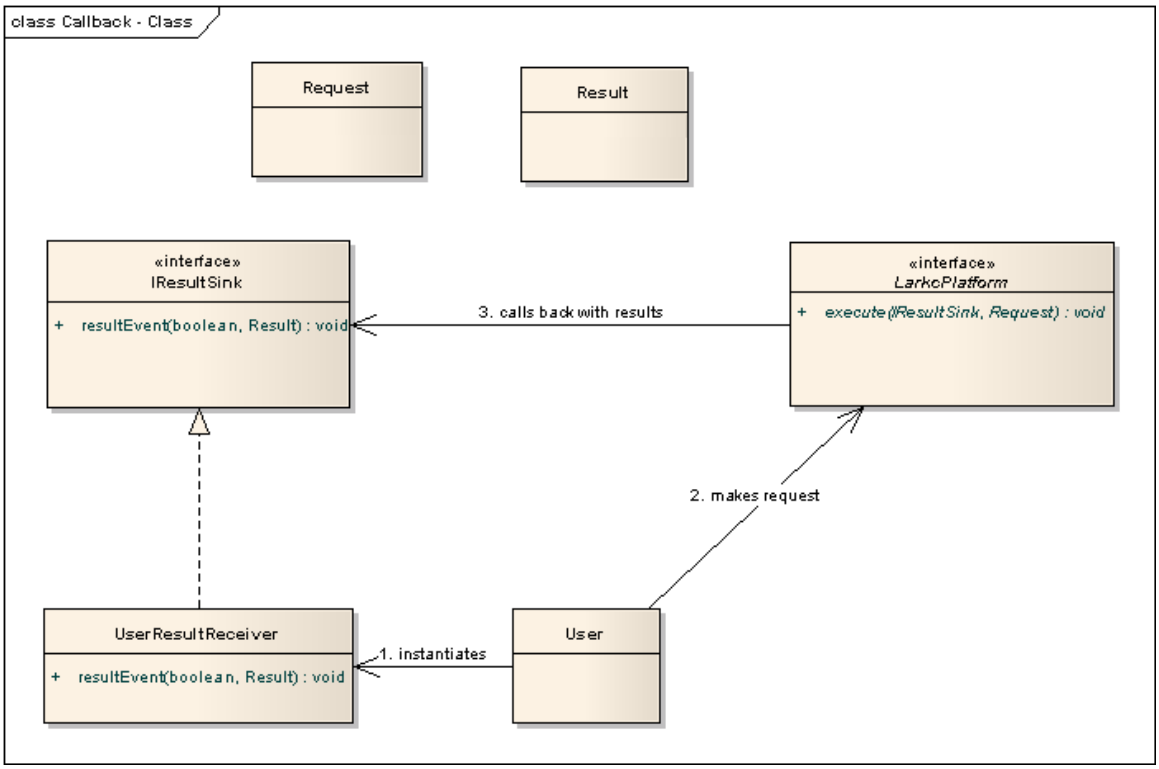

Figure 5.1: The UML Class Diagram that models the callback scenario

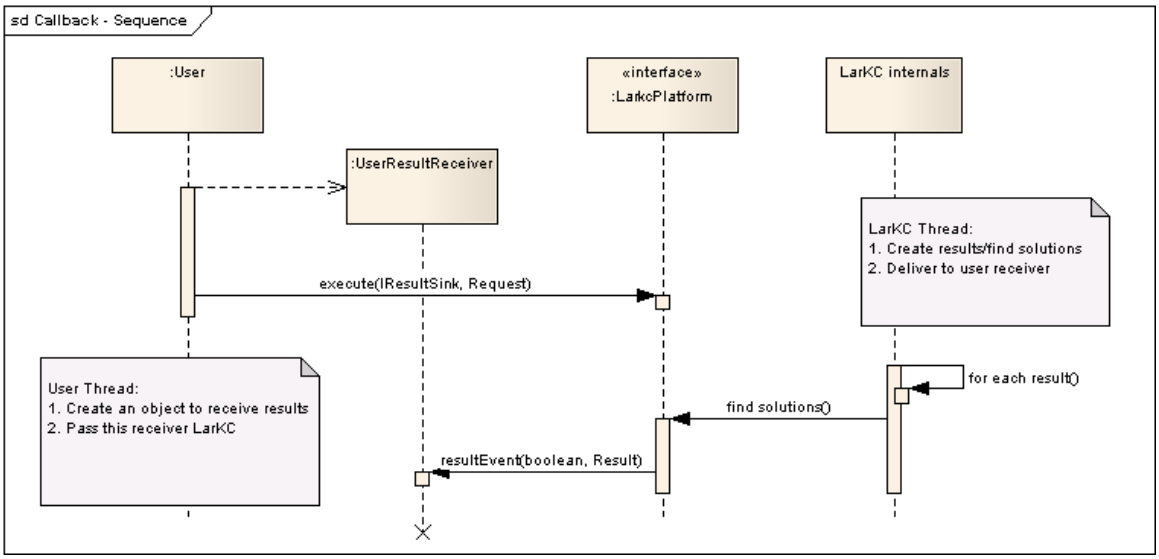

Figure 5.2: Sequence of events for the callback scenario

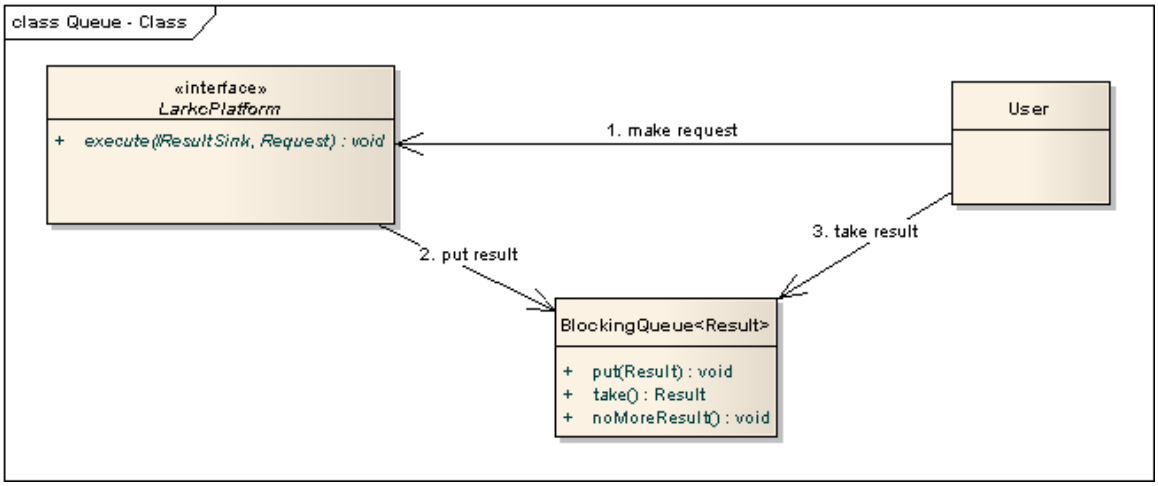

Figure 5.3: The UML Class Diagram that models the Queue-based scenario

- LarKC callback thread executes user code (potentially blocking for unacceptable time).
- The user is required to implement all synchronisation mechanisms when processing results.
- Terminating a search/reasoning task early requires some mechanism to "deregister" the receiver object (with possibly complicated synchronisation issues).
- Results are returned one-at-a-time, although there is the possibility for batch results, i.e. passing many results in one call to resultEvent().

5.1.2 Queue

A common technique for passing messages between communicating threads is to use some form of synchronised queue. In this model, a queue exists at the boundary between the components (or the system and a user). As messages are generated by one component, they are put on to the queue for the target component. Whenever the target component thread is available for processing messages, it "takes" messages from its dedicated queue. Figure 5.3 shows the UML class diagram that models a callback scenario.

All synchronisation in this model is handled by the queue itself. The queue manages contention between threads putting and taking from the queue simultaneously. Also, attempting to take from an empty queue can either block until there is something in the queue (take) or return immediately if the queue is empty (poll).

For LarKC, the messages passed between the platform and the user will be query results. The sequence of events would roughly be like this (the corresponding UML sequence diagram is depicted in Figure 5.4):

1. User submits a query to the platform - this call returns immediately with a (handle or reference to a) result queue (or this could be provided by the user).

2. The platform starts putting results in to the queue as they are generated and continues until there are no more results or it is told to stop.

3. User enters a loop removing from the queue until there are enough results or there are no more results.

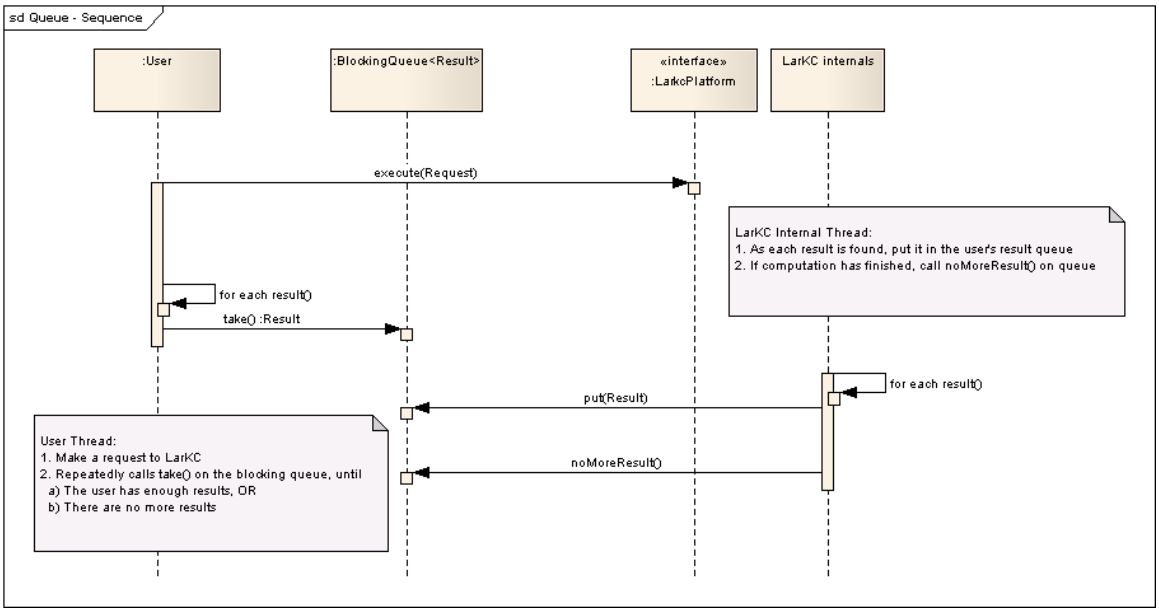

Figure 5.4: Sequence of events for the Queue-based scenario

Advantages of the Queue-based method

- Very simple model.
- Very loose coupling between components.
- LarKC thread executes only LarKC code and user thread executes only user code.
- No explicit synchronisation mechanisms are required - all handled by the queue.
- User code can be essentially single threaded.
- Suitable standard queue implementations already exist (java.util.concurrent).

Disadvantages of the Queue-based method

- Care must be taken to avoid queue explosion - when the user processes results slower than LarKC generates them (can be avoided with a bounded queue, but with the possibility of losing results).
- User thread must either block on the queue when there are no results, or poll the queue in a loop.

6 Storage/Data Layer of the Platform

In the context of software engineering a data layer is responsible for automating the persistence of the system and user information. Thus, an important requirement is the efficient storage and access to data and the related meta-data. The knowledge representation formalism to be adopted by LarKC, which will be reported in Deliverable D1.1.3, uses an extended RDF-based data model with flexible support for named graphs and triplesets. To summarize the rationale behind this backward-compatible extension of the named graphs we can consider a scenario where plug-ins can share large volumes of data, 'passing' this data from one to another via a shared data layer. Technically, one plug-in can pass a reference to a tripleset (part of the data in a dataset stored in a shared repository), which is subject to processing at the next step of the pipeline.

The major functionalities of the data layer can be summarized as follows:

1. Persistence and retrieval of data through a standard API, optimized to deal with huge volume of information (e.g., streaming support);
2. import from (parsing) and export to (serialization) standards formats (e.g. all the popular RDF syntaxes);
3. standard query mechanism and infrastructure (e.g., SPARQL endpoint);
4. sharing reasoning-related metadata (e.g. attaching weights, timestamps or other system information to RDF statements). This meta-data will be accessible only through the storage and retrieval API;
5. possibility to support light-weight reasoning.

Being isolated through a relatively simple API, the data layer allows different modalities of data storage and interchange:

1. Data sharing among multiple plug-ins;
2. local storage, specific to the plug-in;
3. download of an RDF graph from remote host and making it locally accessible through the standard API;
4. providing access to RDF data, stored as local file in a standard syntax.

A Shared/remote repository can reduce communication and computation costs as many reasoning tasks can be reduced to query evaluation. Communication is optimized because the volume of the query results is usually much lower than the volume of data involved in (transferred for) the evaluation. Moreover, a repository component can be distributed or parallelized under a variety of different schemata, while this remains transparent to the platform and the plug-ins. This way the data layer allows for "separation of concerns" (the storage master takes care of storage optimization, decides on distribution strategy, etc.)

The data layer's API is based on the ORDI [1] data model's API and supports the following six formal operations:

[1] `http://www.ontotext.com/ordi/`

- **Adding a new statement:** The operation adds a new statement to the model.
- **Removal of a statement:** This operation retracts a fact from the model including all associations to triplesets. Compared to the named graph model it has a well defined semantics, i.e. if two triples in different named graphs exist only one will be retracted.
- **Assign a statement to a tripleset:** The operation associates a triple from a named graph to a tripleset. If the statement does not exist the statement is not asserted.
- **Un-assign a statement from a tripleset:** This operation removes the association statement from a tripleset, however it leaves the statement in the named graph. If the statement does not exist or it is not associated with the specified tripleset, the operation causes no changes to the model.
- **Retrieval of statements:** This operation retrieves a set of statements and their triplesets based on simple pattern match.

The complete Java-based implementation and documentation of the ORDI data model's API could be found at: `http://ordi.sourceforge.net/apidocs/index.html`

7 Prototype implementation: Baby-LarKC

The design outlined above has been implemented in two test-rigs in order to validate the general LarKC ideas. These will be briefly described here.

7.1 Scripted DECIDE Plug-in

This test-implementation is available at `http://www.larkc.eu/baby-larkc`. This implements a simple LarKC pipeline. The functionality of this pipeline is to act as a SPARQL endpoint to RDF triples available anywhere on the Semantic Web. The pipeline consists of the following plug-ins:

- **QueryTransformer:** It takes the user-input SPARQL query and transforms it into a number of triple-patterns that summarizes the query patterns that make up the users' initial query.
- **Identify:** The triple-pattern queries that have been created are being used to invoke Sindice, which returns URI's at which triples conforming to these patterns can be found. The contract parameter of the Identify plug-in determines how many of these URI's should be returned (streaming fashion).
- **Select:** The current Select plug-in only displays two very simple behaviors, either select all or select only the first URL.
- **Reason:** The Reason plug-in dereferences the triples to be found at the remote URLs, loads them into a local Jena store and executes the original user SPARQL-query on them.
- **Decide:** These plug-ins are under the control of a simple scripted Decide plug-in that executes these plug-ins in the above fixed order, iterating the Identify plug-in with a small value for the contract parameter.

Because of the contract-parameter on the Identify plug-in, this pipeline yields an increasing stream of answers to the original query, as increasingly more URIs are returned by Sindice.

7.2 Self-configuring DECIDE Plug-in

A second version of the above scenario has been implemented but this time with a more intelligent Decide plug-in. Using the Cyc platform, each of the plug-ins (QueryTransformer, Identify, Select, Reason), as well as a number of other plug-ins (e.g. multiple QueryTransformer) register themselves by adding statements in a meta-knowledge-base, stating facts about their type (Identify, Select, etc.) and their I/O signature. Using these signatures, the Decide plug-in decides which potential pipelines can be configured with these plug-ins, then chooses one, and subsequently executes it.

7.3 Discussion

In this section we discuss some of the important aspects of Baby-LarKC.

7.3.1 Remote Execution

The Identify plug-in is written as a stub-plug-in which itself makes remote calls to Sindice (currently running in DERI Galway, Ireland). Writing such stubs is a quick way to obtain remote execution of plug-ins. We intend to write skeleton software to enable parties outside the consortium to make their web-accessible plug-ins available as LarKC plug-ins.

7.3.2 Same code between the two test-rigs

The two test-rigs only differed in their code for the Decider, and in some minimal wrapping code for each plug-in to register the required information about itself in the meta-knowledge-base. All the other code is exactly the same between the two test-rigs, giving us confidence that indeed plug-ins will be re-usable under different Decide plug-ins.

7.3.3 Limited scalability because of data transfer

Obviously the Baby-LarKC pipeline is not massively scalable since it relies on physically loading the selected remote triples into a local Jena store in order to execute the query. This physical transfer is the main bottleneck of this system and the reason why we refer to it as a "Baby" implementation of LarKC. An interesting alternative plug-in would execute the SPARQL query remotely at each of the data-stores that Sindice identifies, or at least do so for all of the data-stores that are available as SPARQL endpoints.

8 Concluding Remarks

In this document we have described the initial operational framework aimed at supporting the development of applications that require massive distributed incomplete reasoning at Web-scale. We presented an overview description of the first version of the LarKC platform in terms of its components (plug-ins) and the functionality provided by the platform. Plug-ins were defined in terms of their interfaces and behaviour. We discussed the three usage phases of the platform and described how the platform supports user interaction. We also introduced and described the datamodel used in the platform.

The description of the platform was complemented by a study of the three use cases considered in the project. This analysis was aimed at identifying the required functionality to be provided by the platform in order to support the implementation of applications in each scenario. Moreover, the analysis served as an input to the definition of the various plug-in types that constitute the architecture.

This document also reported on some important issues related to the design and implementation of the LarKC platform. In particular, we reported on the ongoing discussions in WP1 and WP5 concerning a various architectural aspects such as remote invocation of components, trust and security, data storage and, support for anytime behaviour. Finally, we described a prototype implementation of the current LarKC architecture that implements the design decisions discussed in this document.

The work reported in this document constitutes the first step towards the design of the LarKC platform and therefore many issues still remain open. One such issue is the specification of Quality of Service for the platform and its plug-ins. The definition of the various plug-in types presented in this document was purely concerned with their functional aspect, i.e. the description of the functionality provided by each of them in terms of their input/output. Since LarKC is also about managing resources to achieve approximate and/or anytime behaviour, the interfaces of the different plug-in types (both between platform and plug-ins and between plug-ins) must also capture QoS aspects. In this respect, an analysis of the different use cases considered in LarKC would be the starting point for providing a non-functional description of the platform and its plug-ins. The result of this analysis will be a vocabulary of QoS parameters for each plug-in type as well as a series of configuration parameters used for "customizing" the execution of the platform according to the use cases' needs. Also, related to the specification of QoS is the choice of an appropriate formalism for the specification of these parameters. This as well as other design issues will be addressed in details in future deliverables of the project.

References

[1] IRMOS Consortium. The irmos project: D2.3.1 state of the art irmos technologies. Technical report, 2008.