Instituto de Engenharia de Sistemas e Computadores
Investigação e Desenvolvimento em Lisboa
Laboratório de Sistemas de Língua Falada

NLE-GRID T3
Multi-Component Application Builder

Natural Language Engineering on a Computational Grid
POSC/PLP/60663/2004

— INESC-ID Lisboa Tech. Rep. 33/2008 —

Luís Marujo, Wang Lin, David Martins de Matos
L^2F – Spoken Language Systems Laboratory
INESC ID Lisboa, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
`{ldsm,wlin,david}@l2f.inesc-id.pt`

This report details the restructuring of the original Galinha system, in order to build the web portal for allowing access to modules, applications, and library interfaces in the our system. The interface uses the application server as a bridge between the interface's presentation layer (HTML/JavaScript, at the browser level) and the infrastructure. In this way, users can ignore details about the underlying infrastructure when submitting jobs for execution via a web interface or the portal's application programming interface.

This revision: January 30, 2008

NLE-GRID T3: Multi-Component Application Builder Natural Language Engineering on a Computational Grid POSC/PLP/60663/2004

Luís Marujo, Wang Lin, David Martins de Matos

L^{2}F – Spoken Language Systems Laboratory
INESC ID Lisboa, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{ldsm,wlin,david}@l2f.inesc-id.pt

Abstract. This report details the restructuring of the original Galinha system, in order to build the web portal for allowing access to modules, applications, and library interfaces in the our system. The interface uses the application server as a bridge between the interface's presentation layer (HTML/JavaScript, at the browser level) and the infrastructure. In this way, users can ignore details about the underlying infrastructure when submitting jobs for execution via a web interface or the portal's application programming interface.

Researchers are often presented with the problem of reusing existing resources and tools. These may have been produced in-house or they may be third-party modules. In either case, the task of learning and managing them is not simple: for instance, some tool may be available but may be deemed to hard to reuse for a particular task, causing the redevelopment of a similar tool. This makes application construction more difficult and diverts productive effort to tasks that have already been carried out.

Another critical aspect of the reuse problem is the contact between old tools and new or inexperienced users. The problem here is often in terms of the time required to acquire the necessary expertise to fully and productively use some resource.

To address the above issues, a web-based user interface for building modular applications was developed. This portal has proved to be quite useful in allowing new users and non-specialists to assemble and test complex prototypes: the only requirement is a clear understanding of the meaning of the data used by each module – a requirement much less stringent than understanding the modules themselves and their underlying requirements, such as dependence on distributed computing architectures. By using the portal, and being able to use a simple function, a user can ignore much, if not all, of the complexity associated with particular tools providing the backend support for that function.

This report details the restructuring of the original Galinha system [8], in order to build the web portal for allowing access to modules, applications, and library interfaces in the our system. The interface uses the application server as a bridge between the interface's presentation layer (HTML [20] and JavaScript [2], at the browser level) and the infrastructure.

In addition, the portal's architecture also presents an application programming interface. This interfaces does for programs what the graphical web interface does for human users, i.e., it allows the underlying complexity of the backend systems to be ignored by providing a simple coherent uniform programming interface.

1 Global description

Like the original Galinha system, the current interface simplifies access to modules, applications, and library interfaces: it enables users to access and compose modules using a web browser.

The Galinha (G8) core is one of the portal's key components. It provides the run-time environment for the execution of the various processes within the web application. Moreover, it maintains relevant data for each user, guarantees security levels and manages access control. G8 acts as a bridge between the interface's presentation layer and the execution infrastructure.

Figure 1 shows the overall functional architecture. The upper layer (G8 API) handles requests from client applications (which include the web interface – see below) and communicates through selected middleware with the underlying application back-ends. The rightmost column represents the fact that the G8 layer can also re-export any imported services, thus appearing as any other application back-end. Client applications invoke G8 via web services or AJAX calls [13].

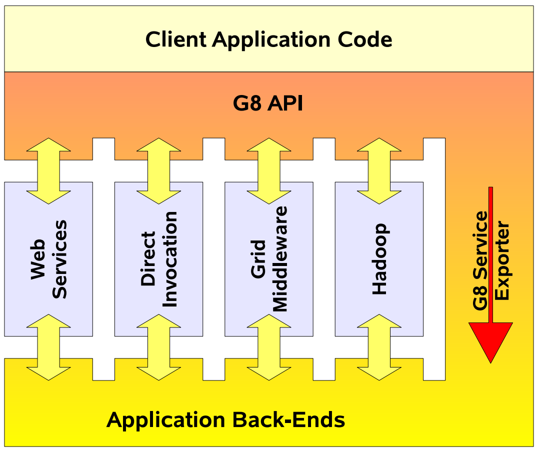


Fig. 1. Functional overview of the architecture.

The following sections detail this architecture: first, in 2, we present the overall G8 internal structure (figure 2), as well as the implementation concerns that governed its creation; then, in 3, we present the structure of the interface and how it communicates with the underlying G8 core; finally, in 4, we present the timed execution daemon used to perform batch operations.

2 The G8 Core

Figure 2 presents a layered overview of the G8 core and the rest of the internal architecture. This architecture implements a three-tiered web application: presentation (see §3), application/logic, and data tiers).

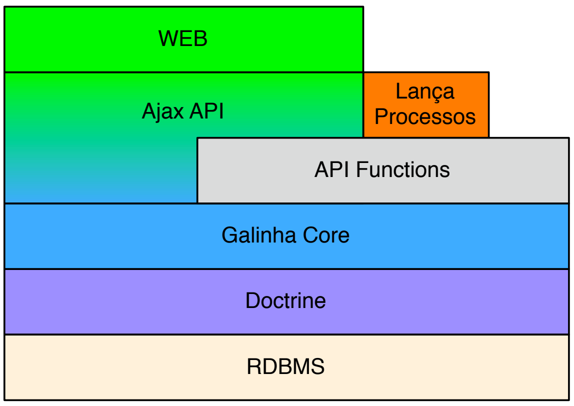


Fig. 2. Layer view of the general architecture.

The application tier is composed by two parts: the internal logic and the interface with the object-to-relation mapping (ORM). The former is responsible for handling all the calls made to the G8 core, the latter is responsible for managing the persistent objects used by the application. Currently, these are the two base concepts for being able to describe operations to be executed: datatypes and the signatures of the operations themselves. Unlike previous implementations of the Galinha system, which implemented a combination of transaction script and table gateway, G8 resorts to an external ORM handler for managing its objects. This allows for cleaner code and sharper definition of the responsibilities in the G8 core and is easier to maintain. On the other hand, it imposes requirements on the communication with the database, even though most ORM engines compensate for that by making the upper layers effectively independent of the storage solution.

The logic layer is responsible for validation and conversion of receive messages and sending the responses to the client interface and for specifying operations to be executed by the auxiliary daemon (§4).

The choice of the ORM engine was guided by the objects that needed to be stored and by the programming language, in our case, PHP [16].

2.1 The ORM solution

Various ORM engines for PHP were studied and are presented below. The implementation status, functionality, and support available for each engine varied widely, from fully functional to alpha-phase frameworks which were unstable, and thus, not reliable in their current state. This was the case of PHPPOS [5] and Junction [3]. The candidate ORMs are presented below.

2.1.1 Lumine

Lumine [6] is a framework that supports multiple database systems, whose schemas are represented in abstract form by XML [21] files, making it attractive solution. However, since development activity seems to be low and little to no support is provided, it was not the selected solution at this time.

2.1.2 Propel

Propel [18] is a stable, well documented framework. It well supported and integrated in the PEAR framework [15]. As Lumine, support for abstract schemas exists (using XML), as well as for multiple database systems. This was the first good candidate, but was ultimately rejected due to the fact that it does not support inheritance and many-to-many associations are not completely functional.

2.1.3 Doctrine

Doctrine [1] implements a data mapper (which is still under development) for PHP the same way Hibernate [] does for Java []. This data mapper provides support for inheritance and inter-object relations, but not for many-to-many relations (in which case there are still some bugs). The data access process is described in PHP unlike the previous options which used XML.

Since this is a project funded by the industry, it is a very active project and progress was observed during the progress of our own project. The documentation is reliable, complete and detailed. Doctrine proved to be superior in every option considered, so it was the select ORM.

Doctrine provides a set of features that make it interesting: a database abstraction layer; a query language (the Doctrine Query Language – DQL); and supports caching (using drivers such as Memcache or APC – Alternative PHP Cache), which allow DQL queries or/and the results to be saved. These caches can be activated both at the connection level or at the connection manager level, being applied just to the connection under consideration or to all connections.

2.2 The Web server

The web server is the actual engine running the G8 core. Since the G8 core was to be available on existing web servers, there were two general options: either G8 would be directly supported by the server, or the web server would have to work as a gateway to the actual engine. Both options were open, but the second was more complex to deploy and the first could provide the necessary features. Although we do not depend on specific features of the Apache Web Server [10], we selected it for providing the support for the G8 core. The reasons go from it being widely used and for the good integration it provides for the PHP engine and deals with low level aspects such as communication concurrency.

2.3 Supported concepts

In the current implementation, types, variables, and operations are supported by the G8 core: types may be created and listed; operations may be created, listed, and executed (and inspected after execution). Operations depend on types.

The logic layer follows the repository or readers/writers architectural style, in which the server generates the information and writes it in the repository (database). Then, processes retrieve the information from the database and process the corresponding operations. At last, the server reads the result from executing the operation and returns it to the client.

Operations can be in one of four states: “under construction”, “awaiting execution”, “executing”, and “executed”. Operations are “under construction” in their initial state, i.e., after being created. When the user executes an operation, a copy with state “awaiting execution” is created. This state will change only after the operation is processed by the daemon: as soon as the operation is processed, the state is changed to the state “executing”. Finally, the operation’s state may change to the “executed” when the execution is over.

This architectural style was adopted due to the inexistence of concurrency programming in PHP, specifically threads, and because the processing time of an operation can the an significant amount of time, which may not be acceptable for the interface.

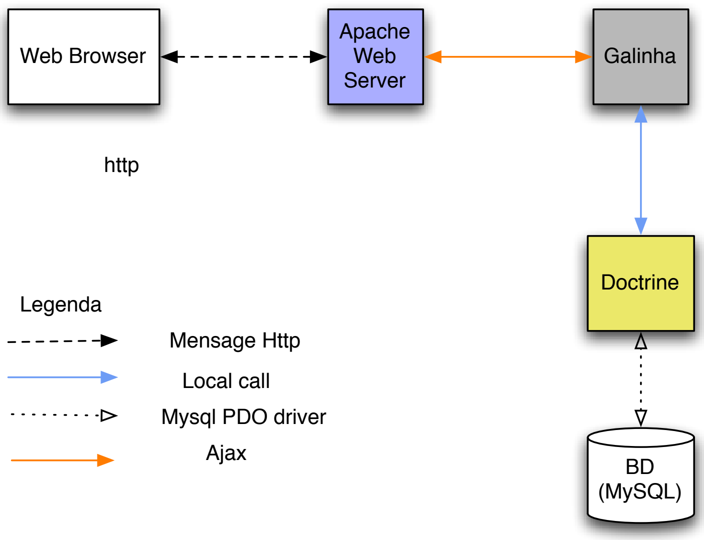


Fig. 3. Component and connector view of the general architecture.

3 Interface

The web interface for the G8 core comprises the presentation tier. In order to simplify deployment and because of its widespread use, the current web interface was build as an extension plugin (implementing a special page) of the MediaWiki engine [11]. The implementation corresponds to a dynamic page responsible for receiving input from the client and sending the request to the server and providing the response to the client via AJAX calls.

The description of the MediaWiki engine is outside the scope of the following report: the description will focus solely on the plugin development of the G8 interface, which follows the use cases defined in §3.1.

3.1 Use Cases

The operations available in the interface are divided into three groups: datatype management; operation management; and chain management (not fully implemented). The type manager is responsible for creation types, editing, and presenting them for inspection. It is capable of handling primitive types, as well as structured types and sequences (arrays). The operation manager is responsible for creating, editing, and listing operations. It is also responsible for preparing operations for execution as well as allowing their results to be inspected. Figures 4 and 5 show the use cases associated with the interface in what concerns types and operations.

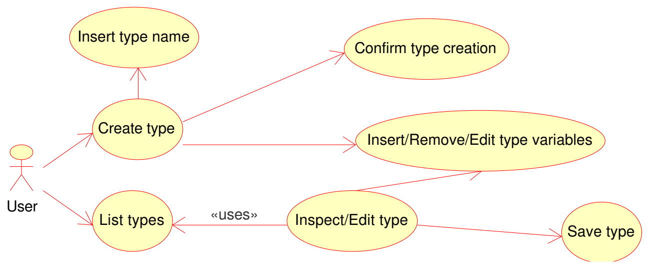


Fig. 4. Use cases associated with types.

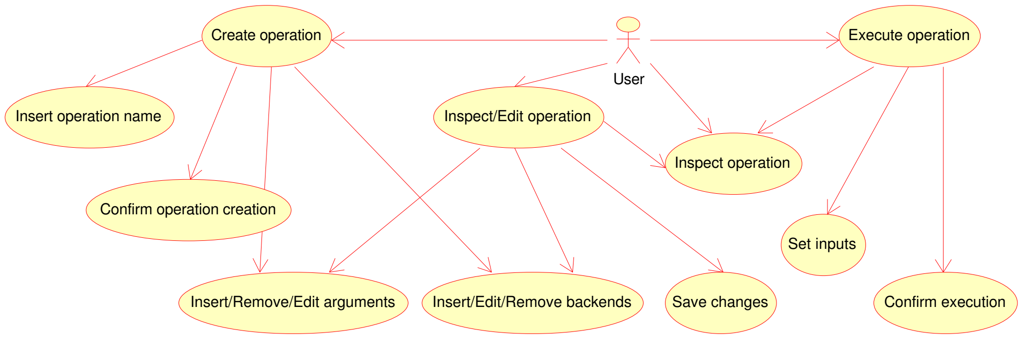


Fig. 5. Use cases for operations.

Not all of the functionality mentioned above was implemented: only the core functionality is available at the time of writing (development continues, though): all operations on types (except support for sequences); all operations on operations (arguments are supported only with primitive types).

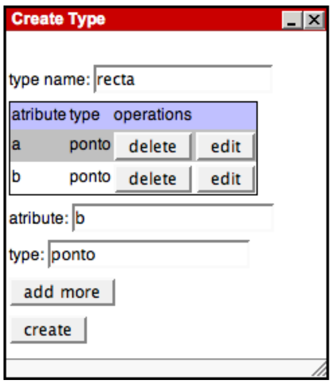
3.2 Interface Design

The following subsections detail the interfaces corresponding to the implemented interfaces.

3.2.1 Creating and Listing Datatypes

Figure 6 shows the interfaces used to create and list types.

To create a type, users are required to do the following: (1) click the "create type" option in the Types menu to open the corresponding interface; (2) fill the "name" field with the type's name; (3) insert the


Create Type

type name: recta

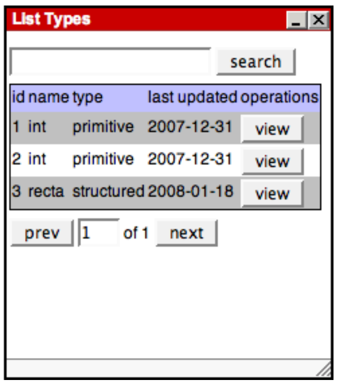
atribute	type	operations	
a	ponto	delete	edit
b	ponto	delete	edit

atribute: b

type: ponto

add more

create


List Types

search

id	name	type	last updated	operations
1	int	primitive	2007-12-31	view
2	int	primitive	2007-12-31	view
3	recta	structured	2008-01-18	view

prev 1 of 1 next

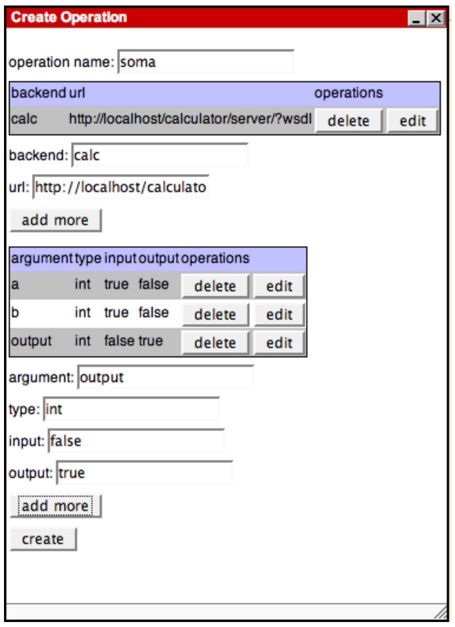
Fig. 6. Interfaces for creating (left) and listing (right) types.

attributes of the type, which are the components of that type: in order to do so, the “attribute” field must be filled with the name of the attribute, the “type” field must be filled with the type of the attribute, and, finally, the “add more” button must be clicked (to successfully create a type, at least one attribute must be inserted and all the attribute’s types must exist); and (4) confirm the type’s creation by clicking the “create” button.

The listing of types allows users to filter the result list by providing a search string and clicking the “search” button; edit/view a type by clicking the “view” button; switch pages (corresponding to different types) by either clicking the “prev” and “next” buttons or by changing the page number: in the latter case, if the page is invalid it will be ignored.

3.2.2 Creating and Inspecting Operations

Figure 7 shows the interfaces used to create and inspect operations.


Create Operation

operation name: soma

backend	url	operations	
calc	http://localhost/calculator/server/?wsdl	delete	edit

backend: calc

url: http://localhost/calculato

add more

argument	type	input	output	operations	
a	int	true	false	delete	edit
b	int	true	false	delete	edit
output	int	false	true	delete	edit

argument: output

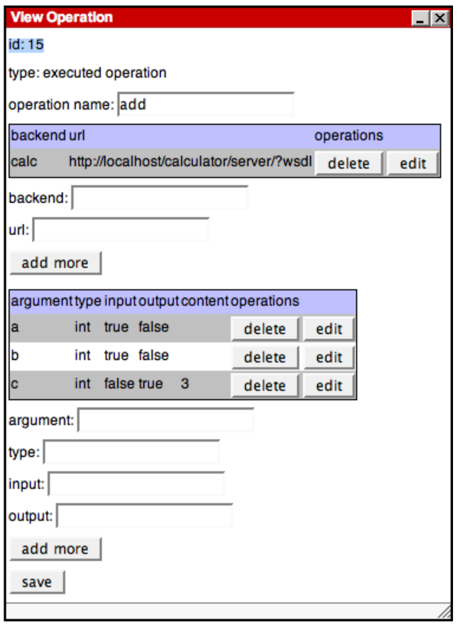
type: int

input: false

output: true

add more

create


View Operation

id: 15

type: executed operation

operation name: add

backend	url	operations	
calc	http://localhost/calculator/server/?wsdl	delete	edit

backend:

url:

add more

argument	type	input	output	content	operations	
a	int	true	false		delete	edit
b	int	true	false		delete	edit
c	int	false	true	3	delete	edit

argument:

type:

input:

output:

add more

save

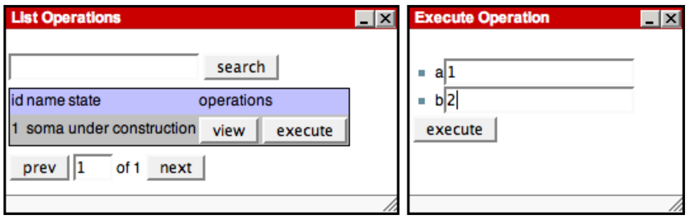
Fig. 7. Interfaces for creating (left) and inspecting (right) operation.

To create an operation, the user is required to do the following: (1) select the “create operation” option in the Operations menu to open the corresponding interface; (2) fill the “name” field with the name of the new operation (this name must be the same as the name of the operation in the backend’s WSDL[1] [22]) description; (3) insert the at least one of the operation’s backends, which are the locations (currently, only the first is used) the operation will be executing on (fill the “backend” field with the backend’s name, the “url” field with the backend’s URL, and click the “add more” button); (5) insert the operation’s arguments, which are the parameters that will be passed to the backend function (fill the “argument” field with the argument’s name, the “type” field with the argument’s type, the direction of data flow: `true` for input, or `false`, for output, and click the “add more” button); and (6) confirm the operation’s creation by clicking the “create” button.

3.2.3 Listing and Executing Operations

Figure 8 shows the interfaces used to list (left) and execute (right) operations.

The list interface provides the following functionality: filtering the operations list by providing a search string and “search”; editing/inspecting an operation by clicking the “view” button; executing an operation by clicking the “execute” button (opens the execution interface); and navigate operations by using the “prev” and “next” buttons or by changing the page number.



id	name	state	operations
1	soma	under construction	view execute

a 1
b 2

Fig. 8. Interface list and execute operation

Executing the operation is as simple as filling the available input fields and clicking the “execute” button. If an error occurs, a message will notify the user, indicating the probable cause: the name of the operation is different from the one specified in the backend’s WSDL description; the backend does not exit; the URL does not exist.

3.3 Interface Implementation

As mentioned above, the interface is implemented as an extension plugin of the MediaWiki engine. It was chosen because it is easy to configure and is useful to save any type of information about the system, namely the documentation. The wiki page which communicates with the G8 core is a special page named “GalinhaInterfaceImproved”. Besides the HTML/JavaScript code, the page also uses two libraries to produce a richer interface by providing a menu and dragable windows: the “COOLjsMenu” and the “windowfiles” libraries. Finally, to facilitate the use of the AJAX framework, the Sajax toolkit was used [4].

The JavaScript functions needed by the interface are generated by PHP code in the Ginterface package, which comprises the classes shown in table 1.

3.3.1 Other tools considered in the creating of the interface

Other options for building the interface were considered, and may yet be used. Nevertheless, they were not used in the prototype web interface.

Google Web Toolkit (GWT) [14] is a framework used to build dynamic web applications. In this framework the programmer codifies the frontend using the Java language and the interface itself is generated au-

[1] Web Services Description Language.

Class	**Function**
`GInterface`	Encapsulates the common behavior of every interface (currently, this is only the state and the used drag-able window)
`GListInterface`	Encapsulates the common behavior of listing interfaces
`GCreateInterface`	Encapsulates the common behavior of creation interfaces
`GViewInterface`	Encapsulates the common behavior of editing/viewing interfaces
`GListTypesInterface`	Generates the interface to lists types
`GListOperationsInterface`	Generates the interface to lists operations
`GCreateTypeInterface`	Generates the interface to create types
`GCreateOperation`	Generates the interface to create operations
`GViewTypeInterface`	Generates the interface to view types
`GViewOperationInterface`	Generates the interface to view types
`GExecuteOperationInterface`	Generates the interface to execute operations

Table 1. Classes in the Ginterface package.

tomatically. The reason this tool was rejected was due to the fact that it would be troublesome to integrate the generated code with MediaWiki and Sajax.

Yahoo User Interface Library (YUI) [17] is a library made by Yahoo with many features to make rich interfaces. These include drag and drop, event manager, forms and more. This option was not chosen because of the fact it does not provide an essential part of the interface, drag-able windows.

Qooxdoo [7] is a framework based in JavaScript classes(pseudo classes) that produces rich interfaces. One of the big appeals in this framework is that it not only makes developing the interface easier by presenting a lot of pre-done work but it also generates the API documentation from the implementation which is presented in an rich environment itself. This option has both drag-able windows and menus objects. This option was just recently considered so it wasn't as studied as the previous two, but it is a promising option in the future.

Ext-js [9] is another option to be considered. This library started as an extention to the YUI library but as it grew over the time it became a stand-alone library. This option provides an easy integration with other libraries such as YUI and prototype by providing adaptors witch diminishes the size of the source code needed to include.

4 Operation Execution

Operations are not currently directly executed by the G8 core: rather, the core prepares an operation for execution and simply notifies the interface that the operation has been submitted to the execution backend. This backend is the subject of the current section.

The first approach was to build a daemon in C++ for processing the operations requested by the upper levels. However, since we could not find a data mapper capable of loading the objects managed by Doctrine (compatible with Hibernate for Java) for C++, we decided against it. Besides, it would be necessary to repeat the code corresponding to the domain model.

The code was, instead, written in PHP. In this way, it was able to directly use the G8 core API for accessing the necessary entities. A problem remained, however: the first approach envisioned a C++ daemon, with internal activity. The code could be run in PHP, but low-level functions are better handled in C/C++. We decided, thus, for a hybrid approach: keep the execution code in PHP, as a web request handler, and use the cURL library [19] for making timed requests to the web server. Using this approach, the concurrency aspect was also simplified, since it is implicitly handled by the web server. Moreover, since it is easier to handle web request in PHP than in C++, the distribution of responsibilities also makes the code easier to understand and, consequently, to maintain.

The implementation uses a C++ class that, every 0.5 seconds (in the default configuration), requests the G8 launcher (`launcher.php`). A somewhat unrelated precaution had to be taken: since web servers log every request, we had to instruct ours not to log the launcher requests (they do not contain operation-specific information and would produce two lines of useless text every second).

5 Conclusions

Our objectives for this task have been successfully completed. This does not mean that the work is finished: for instance, in the interface, it would be interesting to consider using either the extjs-2.0 library or the qooxdoo framework. This change would make the task of building the interface easier and, at the same time, it would enable it to become richer.

Regarding the topics in the logic layer marked as unimplemented or as undergoing implementation, work is still proceeding and the remaining options will become available. This is the case for the support of complex types and operation chains. In this case, chains will come into the system as extensions of normal operations, following the Composite design pattern [12]. Security issues must also be taken into consideration: this includes validating requests and authenticating/authorizing users. Ultimately, performance issues might call for certain parts of the existing work to be redone.

References

1. Doctrine – Open Source PHP 5 ORM. `http://www.phpdoctrine.org/` (visited January 2008).
2. Javascript dom. `http://www.howtocreate.co.uk/tutorials/javascript/domstructure` (visited January 2008).
3. Junction PHP – A PHP 5 Object Persistence Layer. `http://junctionphp.com/` (visited January 2008).
4. Sajax – Simple AJAX Toolkit. `http://www.modernmethod.com/sajax/` (visited January 2008).
5. PHPPOS, 2003. `http://phpos.si.kz/`.
6. Lumine, January 2008. - `http://www.hufersil.com.br/lumine/enduser/index.php`.
7. 1&1 Internet AG. qooxdoo. `http://qooxdoo.org/` (visited January 2008).
8. David Martins de Matos, Joana L. Paulo, and Nuno J. Mamede. Managing Linguistic Resources and Tools. In N. J. Mamede, J. Baptista, I. Trancoso, and M. das Gracas Volpe Nunes, editors, *Computational Processing of the Portuguese Language – Proc. of the 6th Intl. Workshop, PROPOR 2003*, Lecture Notes in Artificial Inteligence, pages 135–142, Faro, Portugal, June 26–27 2003. Springer-Verlag, Heidelberg. LNAI 2721 – ISBN 3-540-40436-8.
9. Ext JS, LLC. Ext JS – JavaScript Library. `http://extjs.com/` (visited January 2008).
10. The Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org/` (visited January 2008).
11. Wikimedia Foundation. *MediaWiki*. Wikimedia Foundation. `http://www.mediawiki.org/` (visited January 2008).
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. ISBN 0-201-63361-2.
13. Jesse James Garrett. *Ajax: A New Approach to Web Applications*. Adaptive Path, LLC, San Francisco, California, February 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.
14. Google. Google Web Toolkit. `http://code.google.com/webtoolkit/` (visited January 2008).
15. The PHP Group. PEAR – PHP Extension and Application Repository. `http://pear.php.net/` (visited January 2008).
16. The PHP Group. PHP: Hypertext Preprocessor. `http://www.php.net/` (visited January 2008).
17. Yahoo! Inc. The Yahoo! User Interface Library (YUI). `http://developer.yahoo.com/yui/` (visited January 2008).
18. The Propel Project. Propel, January 2008. `http://propel.phpdb.org/trac/wiki/Users/Introduction` (visited January 2008).
19. Daniel Stenberg. cURL. `http://curl.haxx.se/` (visited January 2008).
20. W3C. *HTML – HyperText Markup Language*. World Wide Web Consortium (W3C). `http://www.w3.org/` (visited January 2008).
21. W3C. *Extensible Markup Language*. World Wide Web Consortium (W3C), 2001. `http://www.w3.org/XML/`.
22. World Wide Web Consortium (W3C). *Web Services Description Language (WSDL) 1.1*, March 2001. `http://www.w3.org/TR/wsdl`.