

PROGRESS Portal Access Whitepaper

Maciej Bogdanski, Michał Kosiedowski, Cezary Mazurek, Marzena Rabięga, Małgorzata Wolniewicz
Poznan Supercomputing and Networking Center

April 15, 2004

1 Introduction

The PROGRESS Grid Access Environment which came as a result of the grant co-funded by the Polish State Committee for Scientific Research and Sun Microsystems Poland, has been designed to facilitate the access to grid resources and services and facilitate the work required to build grid user interfaces. The portal access to the PROGRESS Grid Environment is handled by two separate modules: the Grid Service Provider and the Portlets, which constitute the content of the PROGRESS HPC Portal. Both those modules come as a part of the PROGRESS grid-portal environment, which is presented in Figure 1. The designed environment is aimed to deliver resources belonging to the PROGRESS grid infrastructure.

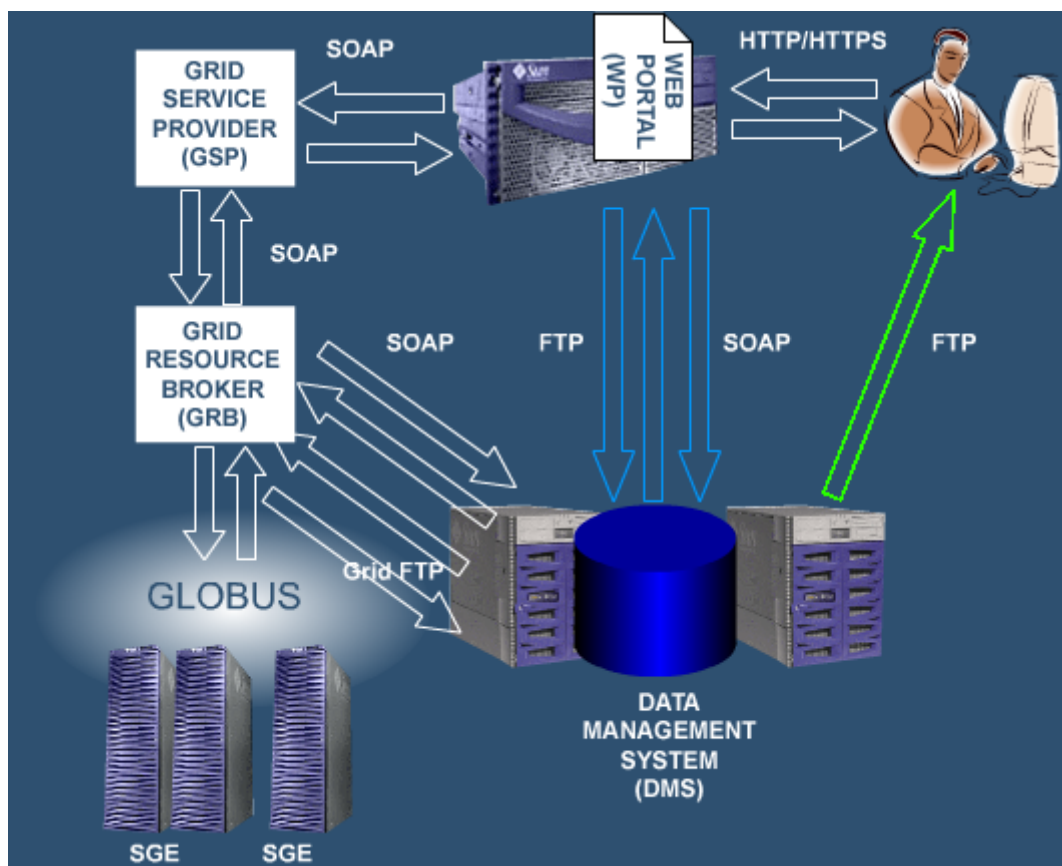


Figure 1 PROGRESS Grid-Portal Environment

The PROGRESS grid infrastructure accessed by the tools designed and deployed within the scope of the PROGRESS project consists of three Sun computing servers: two in

Poznań and one in Kraków. They are connected via dedicated channels of the PIONIER research network. The computing resources are managed by a Globus Toolkit installation, and are delivered by the PROGRESS Grid Resource Broker (GRB).

The GRB is contacted by the PROGRESS Grid Service Provider whenever a user requests a computing job to be submitted for execution in the grid. The GRB decides where and how to run the application associated with the job based on the job requirements and the current status of the grid resources. The job description is passed in a form of a specially designed XML language: XRSL.

The grid services grouped into the GSP are available for use within the PROGRESS HPC Portal, which is the prime user interface; the alternative is a migrating desktop kind of interface. The whole infrastructure is assisted by the Data Management System (DMS), which is used as the source of input data and the destination for results of computing experiments performed in the PROGRESS grid.

2 PROGRESS solution

According to our knowledge at the PROGRESS project design phase, grid-portal environments that were deployed around the world at that time were implemented as 3-tier systems with HPC resources at the bottom, grid management systems (GMS) in the middle and portals in the top tier (see Figure 2). This architecture works, but seems to be not flexible enough. In such a case when creating a new portal administrators are required to build both – the presentation and logical layers of the environment. Thus, there are as many installations of the logical layer as there are portals. This approach is also not very useful for business and market applications. The PROGRESS grid-portal environment which is available from PSNC introduces a new solution to this problem. Functions of the logical layer of the portal are grouped into a separated module running independently of the other modules in the environment. This module is called a grid service provider (GSP) and adds another tier to the system, right above the GMS and below the portal itself. The GSP works as a web services factory. The portal in the new architecture serves as the user interface and implements just the presentation functions.

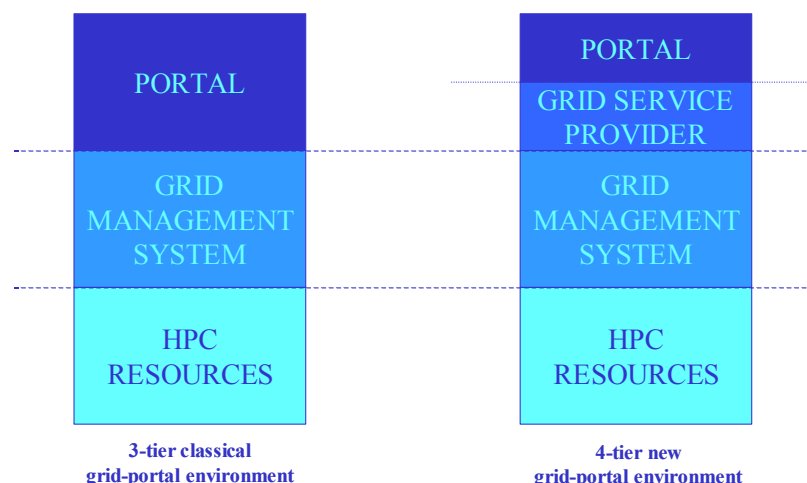


Figure 2 Grid-portal environment architectures

Using a GSP within the grid access environment makes the use of the grid resources more comfortable to the end users. The GSP serves as the source of grid resources: computing power, grid enabled applications, collaboration tools and others, to web portal and other user

interfaces. Installed on a separated server connected to the grid access network the web services based GSP allows for easy building of numerous portals and other user interfaces utilizing the same grid services. Applications coming from one application repository can be accessed with the use of two or more different interfaces. What is more, it is easy to imagine creating various thematic web portals: a bioX portal, an electric engineering portal and a physics portal. Those thematic web portals may also use the same grid services and resources available through the GSP. The next advantage is the possibility of providing all GSP clients with computing resources belonging to two or more different grids. All these resources can again be available through one and the same module, the GSP, and can again be easily utilized by the computing web portals.

3 Grid Service Provider

The PROGRESS GSP is a web services factory with J2EE business logic applied upon the relational database system. It provides four services:

- the Job Submission Service(JS)
- the Application Management Service (AM)
- the Provider Management Service (PM)
- plus an example of an informational service, the news service, intended for use by web portals.

Each of these services comes as a separate EJB application with a Web Services interface.

The JS service delivers functions for computing job building, submitting them to the grid for execution and viewing the results. Methods for building jobs allow for creating jobs, adding and editing job tasks, setting values of application arguments, providing references to job input and output files, and setting the required grid environment.

The JS prepares a job description using the XRSL language. The description is then transferred to the GRB for the execution of the job. When a job is active its current execution status can be monitored by contacting the GRB in request of the execution information. Additionally, the GRB reports back on grid events connected with the execution of the job, like activating a job task, success of the execution or a failure. Finally, the results of the job are available through obtaining references to output files.

The JS supports both: jobs with a single execution of an application and workflow jobs. Each workflow can contain several tasks in any configuration. Tasks may be organized in sequences and parallels. It is not only possible to build workflowed grid jobs using the PROGRESS JS service, but also to create a job using a so called virtual application. Virtual applications are in fact applications consisting of several (or more) components. Each component is an application available for execution in a single-task job and the virtual application descriptor contains information on the structure of workflow to be performed. A task descriptor, relating either to the task of a single-task job or a task of a sequenced or paralleled one, contains a reference to an application. Each task descriptor references exactly one application and the reference comes from the application factory managed by the AM service.

As it was already mentioned, the AM service manages the PROGRESS application factory. An application descriptor contains a reference to the application executable (except when the application is a virtual application). It is either a reference to a file stored in the DMS or a path to a binary on grid computing server filesystems. Each application is also described by a set of available (required or optional) arguments, required environment variables and required input and output files. Application arguments, environment variables, and additionally its grid resource requirements, may all be assigned values. These values may be different for different application configurations. Application configurations are stored and

viewed as independent applications. One executable may be referenced in as many applications (or application configurations) as required.

The provider management service enables keeping up-to-date information on services available in the GSP. The GSP services are web services therefore their descriptors include information on the URL at which the service is available, the service namespace reference (URN) and the service WSDL reference. Some PROGRESS GSP services may have multiple instances. If a service can have multiple instances, it is allowed to create, remove and manage its own instances. Each instance of a service may be understood as a separate virtual resource within the service with its own space in the service database. An example of an instance enabled service within the PROGRESS GSP is the news service.

Currently, the GSP package is deployed and has been tested within J2EE RI 1.3. The package, including the Web Services interface, was built with the use of the Forte For Java 4 IDE. The next version will, however, feature a WS interface based on Apache's Axis toolkit.

4 Portlets

The Portlets are the base to create a web portal accessing the services provided by the GSP, and the DMS. The PROGRESS HPC Portal currently features 5 portlets:

- "My computing jobs": This portlet enables the creation, configuration and execution of grid jobs in the grid, as well as downloading and visualizing the results. It utilizes the functionality of the GSP's job submission and application management services and the DMS's data broker.
- "Applications": This portlet enables the management of grid-enabled applications within the GSP application repository. It co-operates with the GSP's application management service and the DMS's data broker.
- "My data": This portlet enables the management of the files stored within the DMS. It also provides the possibility to upload and download the files from the DMS. The "My data" portlet communicates with both DMS's brokers.
- "Management": This portlet enables the management of a GSP instance. It uses methods provided by the GSP's provider management service and methods intended for the management of service instances, which are provided by GSP services.
- "News": This portlet provides a service for short news presentation and edition. It utilizes the short news service available within the GSP package.

The currently deployed portlets were built on the base of the test client applications automatically generated by the FFJ (FFJ 4?). With the use of the tool enabling automatic generation of this client application we built client proxy classes to enable access to the developed applications. These classes were slightly adjusted to facilitate their usage within our client applications. We simply replaced the full list of methods corresponding to the WS methods of a service with a few *executeMethod()* methods allowing invocation of any method of the remote service.

The DMS features two external services with WS interfaces: the data broker and the metadata broker. These brokers form the access point to the Data Management System and manage all of the client requests. The DMS Web Services, including the data and metadata brokers' interfaces, have been implemented using the Apache SOAP implementation. To deliver the functionality of the DMS services to the portlets developed for use within the PROGRESS HPC Portal, we built proxy classes similar to those prepared for the GSP services. It allowed communication with the brokers and delivering their functionality to the PROGRESS portlets. The PROGRESS portlets utilize the data broker service.

The adjusted proxy classes allowed us to build portlets which constitute the PROGRESS HPC Portal. However, these classes did not provide the functionality to improve the process of creating new portlets utilizing the WS services. They delivered the data obtained from the services in a form of objects, which were hard to manage and efficiently merge to create an output HTML document. Even though the WS responses could be returned as XML documents, they were difficult to manage when it came to combining the responses after invoking multiple WS methods during one user request. It was also not efficient enough when it came to building more than one portlet communicating with the same WS service. In such a case each of these portlets must have managed the proxy class instances on their own and inside their own source code. This resulted in a software package with an unclear structure, which was difficult to manage and reuse.

To improve the efficiency of building portlets utilizing the functionality of GSP and DMS services, we decided to design the PROGRESS Portlet Framework. It allows preparing a higher-level interface to Web Services with special beans to store the data obtained from WS responses and classes used to translate HTTP requests into the invocation of proper WS methods. The latter additionally help to quickly build new portlets on top of them to utilize the already existing functionality that they provide.

The PROGRESS Portlet Framework features five layers in its architecture and forms a clear structure of an application accessing Web Services. The bottom layer consists of *Web Service Proxy* classes, which handle the SOAP communication with WS services. Each of these classes is accompanied with a *Web Service Invoker* class to abstract the WS method invocation and form the second layer. The middle layer features *Request Handler* classes, which are capable of translating user's requests into a proper stream of invocations of WS methods. The fourth layer provides *Content Generator* classes, which prepare the content to be returned to the user. Finally, the most upper layer includes *Provider* classes, which can take a form of portlets. The layered architecture of the PROGRESS Portlet Framework is presented in Figure 3.

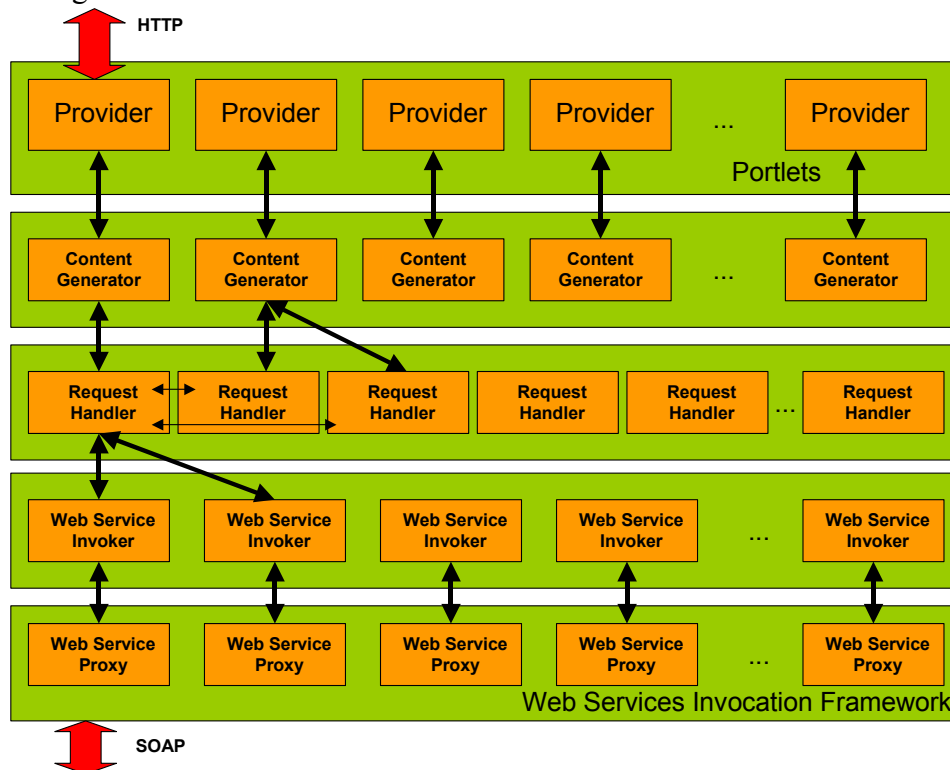


Figure 3 Architecture of the PROGRESS Portlet Framework

The Web Service Proxies extend a base proxy class, which utilizes the functionality of the Web Service Invocation Framework and uses the binding classes generated with the Axis toolkit based on the WSDL descriptions of services. In the PROGRESS Portlet Framework we decided to replace the old proxy classes generated by the Sun's IDE with the ones built with the use of WSIF and Axis, because the Sun's IDE is first of all a commercial product, not enabling us to freely use newer versions of the software; secondly, the WS proxy class structure produced by Sun's IDE changes in later versions of the IDE; and, additionally, the generated Web Services are not fully compliant with the WS standard.

The Proxy classes are capable of communicating with the services using the SOAP protocol. They expose the *executeMethod()* method to enable the invocation of WS methods. This method has two required arguments: the name of the method to be invoked and a list of arguments, and two optional arguments: *inputName* and *outputName*, which determine the proper WS method to be executed if the service features more than one method with the same name.

The Proxies' *executeMethod()* methods are invoked by the Web Service Invokers. These classes feature lists of methods corresponding to all methods of the underlying WS. These methods allow performing required actions on the services and return the WS responses in the form of specialized beans, that is, *Page Beans*; we discuss the function of Page Beans further on. The introduction of Web Service Invokers makes the layers responsible for communication with the services clearer in structure and more flexible in use, additionally reducing the number of source code lines needed in the upper layers to prepare a request with a WS method invocation.

In the PROGRESS Portal Framework each Web Service is described in an XML configuration file. The WS configuration files contain basic information about the WS services, including the location of the WSDL file, and the name and namespace of the port type. They are used to initialize the Web Service factory. The WS factory provides simple *Web Service* objects, which are later used to create the WS Proxies.

The Request Handlers are used to invoke proper WS methods based on user requests. One Request Handler method can invoke multiple WS methods. Also, one Request Handler can invoke a set of methods belonging to one or more other Request Handlers. It is very useful to simply include the already existing and implemented streams of WS method invocations as a substream of a stream of methods performed by a new Request Handler. The methods of Request Handler classes are invoked using the Java reflection mechanism. An invocation stream of Request Handler methods (these can be methods of one or of multiple Request Handlers) constitutes the action to be performed as stated in the user request. Each Http Request contains the *action* and *page* parameters, which are analyzed by a Content Generator. Each page is usually combined with a default action used to generate its content; in such case the *action* parameter is not required in the Http Request. Actions can be independent of the requested page or replace the default page action. Actions are performed before the page is generated. Actions and pages are described in XML configuration files, in *actions.xml* and *pages.xml*, respectively. The configurations are dynamically loaded by classes referenced by Content Generators. This mechanism of dynamic load of action and page configurations allows to quickly add new pages to a portlet. When all methods necessary to deliver the content of a new page are available within the already existing Request Handlers, it is enough to add proper lines to the respective configuration files and the required actions are performed on the services, and the new page is generated on the basis of the responses returned by Request Handlers. There is no need to write any lines of Java code for such a page to be generated.

Content Generators are responsible for the preparation of the content returned by a portlet filling in a portal channel. As it was already mentioned, the WS responses are returned by Web Service Invokers to Request Handlers in a form of Page Beans. The data stored within these beans can be easily transformed into *DOM* objects. With every user request, and thus with every action to perform, Content Generators build a Page object in the process shown in Figure 4. A Page contains a DOM, which is built of smaller DOMs representing each of the Page Beans returned by Request Handlers used to perform the requested action. The DOM stored within the Page is used to transform the obtained data into the final content format, for example into HTML.

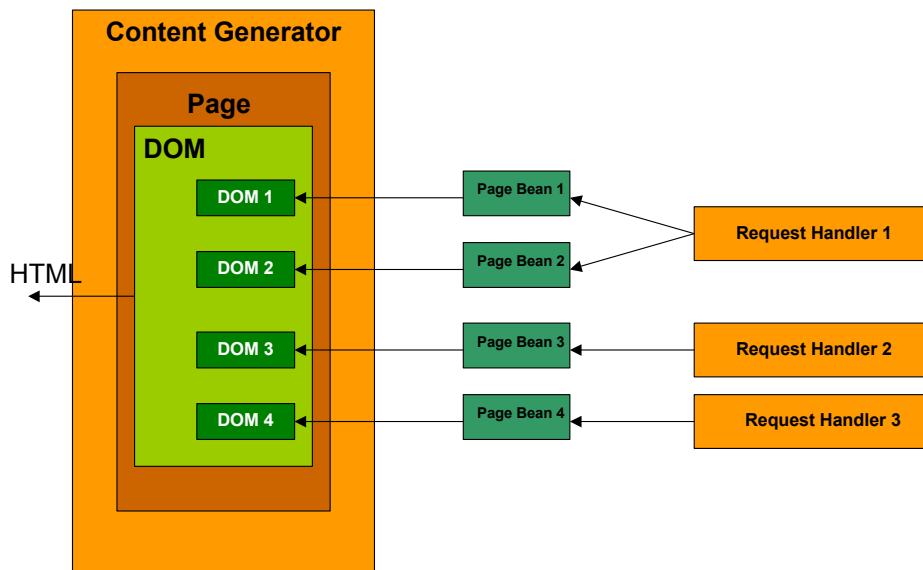


Figure 4 The process of portal channel content generation

Each Content Generator is utilized by exactly one Provider. Providers, which are usually portlets, can take a form of any class used to generate the content of a channel within a web portal, for example the JSR 168 *Portlet* or the Sun's *ProfileProviderAdapter*, or even a generic *HttpServlet*. Providers are the top layer classes, which allow plugging the built web applications into a portal framework and the portal itself.

The layered architecture of the PROGRESS Portlet Framework enables flexible and easy development of user interfaces to programmatic Web Services. Its structure is clear to the developer thus facilitating his/her work. The parts of the already existing code are easily reusable when a newly designed portlet is required to utilize the same functionality of the WS services that were utilized in previously developed portlets. The five-layer structure of the Framework also provides a wide range of extensibility. It is easy to extend or replace any of the layers with new objects to adjust its functionality and to make it work with other protocols and data formats.

The PROGRESS Portlet Framework has been used to develop the newest release of the PROGRESS Portlets. They are currently under tests (together with the newest release of the Grid Service Provider which features the Axis WS interface) and will be published in the form of an open source package in the upcoming weeks. All new portlets, that is the specialized application portlets intended to facilitate the use of specific applications available within the PROGRESS grid-portal environment (e.g. a portlet handling Gaussian 98) are developed with the use of the PROGRESS Portlet Framework.

5 Remarks

5.1 PROGRESS packages requirements

5.1.1 Grid Service Provider

- The GSP package has been deployed in the environment of J2EE RI 1.3.1 application server. The package contains standard J2EE applications (one per each GSP service), which means it should be possible to deploy this onto any application server that supports the J2EE 1.3.1 or later specification (for example the Sun ONE Application Server 7).
- The GSP services were prepared as EJB applications and as such require access to a database system. The entity beans are using Oracle 9 as the database system. It should, however, be possible to use any other relational database system, provided that there is a JDBC driver for that database system available. The requirement here would be to replace all Oracle-like SQL queries in the entity beans configuration and some changes involving generation of identifiers for objects stored in database tables; the identifier generation currently relies on the Oracle sequences.
- The package is deployed within the PROGRESS grid-portal environment on a Sun Fire SV 880 machine.

5.1.2 Portlets

- The PROGRESS Portlet Framework enables to easily adjust the developed portlets to any API provided this API involves utilization of the standard *HttpServletRequest*. The Java class implementing the top level interface for a single portlet MUST have access to the *HttpServletRequest* instance corresponding to the user's request to this portlet's content within a portal. The Portlets are being tested in two forms: as separate *HttpServlets* and Sun ONE Portal Server's *ProfileProviderAdapters*.
- The runtime environment of the PROGRESS HPC Portal is Sun ONE Portal Server 6. This could be, however, exchanged to any other portal framework supporting Java Servlets (please see remarks on Authentication and Authorization).

5.2 Authentication, Authorization and Single Sign-On

- The PROGRESS grid-portal environment uses the Single Sign-On (SSO) solution provided by the Sun ONE Identity server to identify and authenticate the portal users between the Portal, the GSP services and the DMS's data broker. Each call to a GSP's or DMS's Web Service method contains an identity token assigned to the user session by the Identity Server. The session is shared between the distributed modules: the GSP and DMS validates the token with the Identity Server before performing any actions corresponding to the call incoming from a portlet.
- The SSO solution may be omitted: the portlets may be made to pass the user identifier to the services, in which case the services trust each incoming call. This scheme works, it is, however, less secure and should not be used in an open grid environment, like for example one working within the Internet.
- The authorization of user actions within the GSP is based on communication with an external authorization system. This authorization system has been based on the Resource Access Decision (RAD) specification and is a work of the AGH Cyfronet Krakow, a partner in the State Committee for Scientific Research grant. The GSP is designed in such a way that this external authorization system may be replaced by any other which can fulfill the need to manage the user access rights database and the

access decision. Each GSP service controls user access to resources based on the decision obtained from this external authorization system. There is a simple GSP module, which is contacted by the services, that is responsible for the communication with the authorization system. It is easy to reimplement this module for communication with any other external authorization system.