A "CROSS-TECHNOLOGY" SOFTWARE DEVELOPMENT APPROACH

Stefan Palanchov, Alexander Simeonov, Krassimir Manev

Abstract: *Contemporary web-based software solutions are usually composed of many interoperating applications. Classical approach is the different applications of the solution to be created inside one technology/platform, e.g. Java-technology, .NET-technology, etc. Wide spread technologies/platforms practically discourage (and sometime consciously make impossible) the cooperation with elements of the concurrent technologies/platforms. To make possible the usage of attractive features of one technology/platform in another technology/platform some "cross-technology" approach is necessary. In the paper is discussed the possibility to combine two existing instruments – interoperability protocols and "lifting" of procedures – in order to obtain such cross-technology approach.*

Keywords: *web-solution, interoperability of applications, interoperability protocols, lifted procedures*

ACM Classification Keywords: *D.2 Software Engineering, D.2.11 Languages (interconnection), D.2.12 Interoperability (interface definition)*

Conference: *The paper is selected from Sixth International Conference on Information Research and Applications – i.Tech 2008, Varna, Bulgaria, June-July 2008*

Introduction

In [Maneva, Manev, 2008] different models for development and distribution of software (MDDS) and their role for the efficiency of the developed software products were discussed. Especially it was stressed the *status quo* of the contemporary models for development and distribution of web-based business-oriented software solutions. Many negatives of the existing models were outlined that lead to high costs or to low quality of the implemented web-based solution in medium and small companies, as well as in the state administration. As a result, the necessity of a new model concept was formulated. In conclusion, the following features of such model were identified:

- It has to guarantee the **independence of the user** from the technologies, i.e. the user has to be free to chose for each component of the solution the existing technology that is the best for this component;
- It has to guarantee that the user will obtain a service with a **quality, which is relevant to the paid cost**;
- It will be very good if the model has the **tolerance for the qualification** of the users and to allow them to extend and update solution, etc.

Dependence of the users from the technologies was identified as a crucial element of the existing models. The notion *independence* is not new in the domain of development of software products for the business. For example, one of the main goals of the very popular approach Model Driven Architecture of OMG [OMG, 2008] is to liberate the process of conceptual design of the solution from the technologies of the implementation. It is used by many developers. Following the MDA concept they first design the solution in conceptual (or technology independent) level and than, automatically, semi-automatically or manually, *map* the elements of the solution in a chosen technology/platform.

It is true that MDA gives **some independence** from the technologies. More precisely MDA gives **full independence, but only in the stage of conceptual design**. In the stage of implementation of the conceptual design there are two possibilities. The fist is to implement all applications with one "clean" technology and to make the solution totally dependent from this technology. The second is to use different technologies in the different applications of the solution. If the second possibility is chosen, then some additional efforts will be necessary for homogenization of the interfaces between interoperating applications. The developers rarely do the efforts to develop the homogenization from scratch and usually rely it to corresponding software (*middleware*), making the product dependent of the middleware.

In this paper we will consider the dependences from the technologies, which are generated on the second stage of applying some concepts, similar to MDA. We will try to investigate the possibility to give to the users more independence from the technologies or middleware. The main objective is that each technology has its own positive elements as well as its shortcomings – a single technology, due to different reasons, could not be ideal. We will try to estimate the possibilities to integrate some of the best features of different technologies on the base of homogenization of the interfaces among languages, proposed by these technologies. We will call this a *cross-technology* software development approach.

In the second section of the paper some terminology and necessary basic knowledge are introduced. In the third and fourth section two instruments are considered that are necessary for implementation of our idea. Some advantages and shortcomings of these instruments are stressed. The idea itself is presented in the fifth section. In the last section some conclusions are given.

Basic notions

In this paper we will call *technology* (or *platform*) some general *concept* or *approach* for development of software. Part of the technology or platform are also some preliminary tailored *components* (classes of objects, small program modules – applets, servlets, etc., and even not very large “stand alone” applications) that implement the concept or approach, as well as the corresponding *tools* (programming languages, IDE, API, DB-interfaces, etc.) dedicated to support creation/integration of the software solutions (i.e. Java-technology of SUN, or .NET-technology of Microsoft).

With the term *dependence on the technology* we will denote different kind of limitations that the users have to obey if choose specific technology/platform for creating/integrating the necessary solution. For example, any attempt of the user for appending new functionality to the solution, developed with a specific technology, has to be implemented “within” the technology. Issuing of a new generation of the elements of the technology could lead to necessity of total upgrading of all bought to the moment elements and, probably, reintegrating of the solution. And more, when an element of the technology is of low quality, comparing with the concurrent products with the same purpose, it is very difficult to eliminate this element from the solution and to replace it with a better one.

Following [Ousterhout, 1998] we have to agree that the contemporary web-programming is really *gluing components* (GUI-components, small applications providing content or services, etc.) in a *solution*. The components are usually written in some *system programming language* (like C, C++ or Java) and as a gluing instrument different *scripting languages* (like Unix-shells, Java Script, PHP, Perl, Tcl, Python etc.) are used. It is possible that some components are also written in a scripting language. The opposite, gluing of the solution with a program written in system programming language is rather nonrealistic – these languages are not dedicated to such purpose.

One of the main objectives for the promoted in this paper cross-technology approach is that the components written in different languages (both system and scripting) are not always able to *cooperate in run time*. Developed technologies/platforms resolve this problem with some “inner”, built-in, interoperability of one or more system programming languages with one or more scripting languages (C – Unix-shells, Java – Java Script, etc.). We are proposing a way to achieve such interoperability in a cross-technology level, i.e. among languages for which built-in co-operability mechanism does not exist.

Interoperability protocols

One of the possibilities for achieving run-time interoperability of components is to use some *interoperability protocol.* Examples of such mechanism for achieving interoperability of different processes written in C/C++ that even could work on different computers, is Remote Procedure Calls (RPC), developed for Unix-like operating systems by Sun [Marshal, 1999]. For applying RPC a unique number is assigned to each interoperating program and, for a given program, a unique number is assigned to each procedure inside the program. Call of a remote procedure is made trough a corresponding *RPC-client* (executing function `rpc_call()` in the calling process). Beside the traditional list of parameters of the called procedure, many other parameters have to be provided also – the identification of the host, the unique numbers of the program and of the procedure that is called, etc. The calling program is waiting as usual to obtain the result or some indication that the remote call failed (timed out, for example) and then continues the work.

The RPC-client is responsible for serialization of the given arguments of standard types, i.e. translating them to an inner RPC form, which is suitable for transportation in the net. Serialization of user defined types is responsibility of the user. For this purpose RPC provide a set of standard procedures and the corresponding procedure has to be called for each included in the user defined type variable of standard type. The request is composed following the rules of the protocol, sent to the host and processed by *RPC-server*. The server performs deserialization of arguments, identifying and calling the procedure and serialization of the obtained result, which is sent back to the RPC-client. Finally, the RPC-client performs deserialization of the result and returns it to the calling process.

The system RPC is developed to support communication among processes, the code of which is written in C/C++. There are samples of systems for generating interoperability protocols for other languages, well checked and proved as mentioned RPC – RMI for Java [Java RMI, 2007], RPyC for Python [RPyC, 2007], CORBA, SOAP, etc.) – that could also be used as a model. There is no popular sample of protocol generating system that is able to provide interoperability of components, written in different languages. For achieving interoperability of processes, the code of which is written in different languages, additional efforts will be necessary and we will discuss them below. We will take some existing systems as models and will try to extend the idea to a system, which is able to provide interoperability protocols for processes written in different languages – *homogenization protocols*.

The advantages of such homogenization protocols are obvious. They could be a first step toward obtaining a total independence of the developer from the technologies or platforms. In such way each component of the solution could be created in the most appropriate language, within most favorable technology or platform. Different components could be executed on different machines and even under different operating systems.

Obvious shortcoming of the homogenization protocols will be the significant amount of time, necessary for the execution of the procedure call. Each call is passing through a complex process of structuring and parsing of the request, serialization and deserialization of arguments and results, etc. As a result the additional time could be many times bigger than the time necessary for the local call. That is why such kind of homogenization is inappropriate for relatively small and simple tasks. There is no sense to organize a remote call (especially to a procedure written in another language) for finding the sum of two integers, for example. Homogenization protocols have to be used only for requesting services that could not be obtained locally or could not be obtained in reasonable price.

Some negatives of the approach could be observed on the example of RPC also. The function `rpc_call()` that performs remote call has 8 arguments due to the rules of he protocol – coding of such call could take time and the probability for giving a wrong argument is significant. One of the arguments is the identification number of the program, which contains the called procedure, and, if the system is implemented as in RPC, the developer has to keep in mind large amount of such identification numbers (in RPC the numbers reserved for programmers are from 0x20000000 to 0x3FFFFFFF). And finally, we described above a very simple scheme of RPC. Really, the approach is much more complex and could be used only by very experienced programmers. All these negatives are not generic and could be surmounted in one well planned implementation.

“Lifted” procedures

On the road for achieving cross-technological interoperability of components written in different languages, we will use one other approach too that we will call *lifting of procedures*. As it was mentioned above, for creation and integration of software solution, languages of different level of abstraction are used. The idea is to choose one language as a basic, to create a library of necessary procedures in this language and than to “lift” each procedure to each of the other languages being in use. With “lifting” of the procedure we will denote the process of making basic procedures accessible from programs written in languages different from the basic language. The assumption is that the basic language is of the lowest level among all used languages. That makes the candidate for a basic language almost unique – the C language. The level of C is low enough. C++ and Java, as well as many of the most used scripting languages inherit the syntax of C and are appropriate for the “lifting” process.

Example of a tool for lifting of procedures is the system SWIG [SWIG, 2007]. It is a typical “open source” project developed and maintained by some enthusiasts on voluntary principle. As mentioned on the official page of the system: “SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages.” Nowadays the most popular scripting languages as Perl, PHP, Python, Tcl and Rubby, as well as non-scripting languages as Java, C#, Common Lisp, etc. are supported by SWIG.

The system SWIG was written initially (by Dave Baezley in 1995) for C as a basic language. In 1996 the system was rewritten for C++ as a basic language. Obviously a bit more high level of C++ was not suitable for the goals of the system because since version 3.1 the software migrated back to C. This confirms our observations that the basic language has to be of as low level as possible.

The idea of lifted procedures originally was to ameliorate the performance of programs written in a specific scripting language, providing a mechanism for calling procedures written in the language C, through which the scripting language was interpreted. The idea happens to be very helpful and soon some versions for other scripting languages interpreted trough C or C++ were created. Finally, the idea happens to be universal and soon versions for non-scripting languages and, which is more important, genetically not connected to C and C++ appeared too.

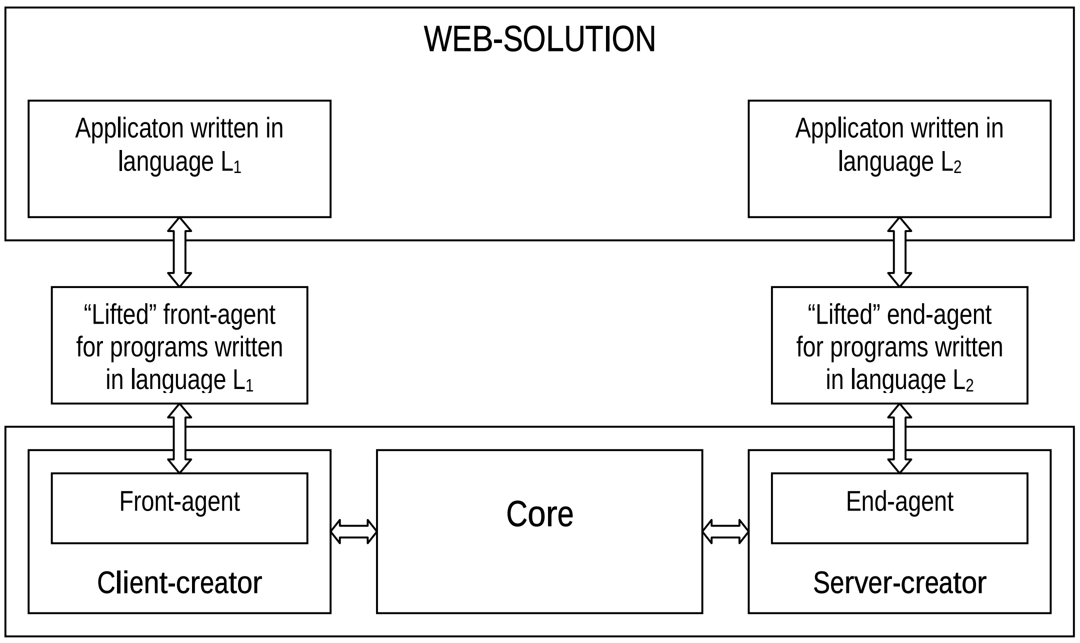
The lifting of procedures practically has no shortcomings beside the fact that for each new language a specific module of the system has to be created. Fortunately the most of used in the contemporary web-solutions programming languages are supported by the current version of SWIG; the software is relatively wide spread and well tested. A few small shortcomings could appear from the peculiarities of some language that could make one universal lifted procedure inefficient in this language. In such case a specific version of the procedure has to be written for each such language.

There is an objective that we have to keep in mind when plan to use “automatic” tools like SWIG. It is quite possible that such automatic tool does not support all constructions of the basic language. For SWIG and C language this seems not to be true, but for SWIG and C++ such problems exist. Fortunately, the systems like SWIG are with open code. This gives a possibility to qualified users to re-develop some specific modules in order to solve some specific problem and, as a result, to contribute to extension and amelioration of the tool.

“Lifted” interoperability protocols

The general idea of this work is to mix the two approaches – interoperability protocols and “lifted” procedures – in order to obtain interoperability of applications written in different languages. As a beginning, let us take some tool for achieving interoperability through interoperability protocols. It could be RPC or some modification of RPC, if some features of RPC are not suitable for implementation of the idea. It could be some other tool with the same functions also. Finally it is possible, following the model of RPC and another existing system, to create a new tool for implementation of interoperability protocols. Let us call this tool *Basic Interoperability Protocols* (BIP). The language of creating of BIP has to be as low as possible – most probably it will be the C language.

Schematically BIP could be split in three parts (see the Figure) – *BIP-client creator*, *Core* and *BIP-server creator*. The Core part is independent of any used language and is dedicated to provide a transportation mechanism between applications (including applications that are working on different computers). The BIP-client creator and the BIP-server creator are dedicated to enable communication of the Core with calling procedure and called procedure, respectively. Both creators could have a part, which is independent of the used languages. But the essential for them is the part that depends of the language of calling/called procedure. It is very probable that these parts will have more than one version (because of the mentioned above particularities of the used language) but we will prefer to refer them as integral elements of the system – the *front-agent* and the *end-agent*.


WEB-SOLUTION
Applicaton written in language L1
Applicaton written in language L2
"Lifted" front-agent for programs written in language L1
"Lifted" end-agent for programs written in language L2
Front-agent
Client-creator
Core
End-agent
Server-creator

Figure

The front-agent and the end-agent of the BIP system (really they are set of written in C functions) are lifted to the level of all used languages. Initially the lifting could be maid by existing SWIG processors. Some additional *lifters* written in SWIG-style could be created if necessary (for example, when some of the existing SWIG processors are not appropriate for BIP, or when a language not supported by SWIG is used).

Really, lifting of front-agent and end-agent are different kind of processes. It is possible to say that lifting of the front-agent is a classical use of SWIG-like mechanism and lifting of end-agent is an attempt to extend the possibilities of the tool with a new functionality that was not initially presumed. Lifted front-agent has to provide to procedures written in high-level language a possibility to call procedures written in low-level language. In our case these are the procedures of the interoperability protocol that have to transport the remote call to the called procedure. Lifted end-agent practically makes the opposite – provides to procedures written in low-level language a possibility to call procedures written in high-level language. In our case these are the called procedures. It will be more correct to say that the end-agent is "taking down" the level of these procedures.

Because initially SWIG-like mechanisms were not designed for taking down the level of procedures, this process will not be as easy as lifting of the level. Anyway, there are mechanisms for solving the problem and one of them is the "callback" mechanism. For some reasons, different of discussed in this paper, the callback of procedures was implemented in SWIG and, even if it is not working very smoothly (see for example [SWIG, 2002]), it could be used for our purposes.

Conclusions

The discussions and the innovative ideas presented in this paper are results of some experiments. The current versions of RPC, as a generator of interoperability protocols, and SWIG, as an instrument for lifting of procedures, were used. First, some experiments with SWIG system were made. Procedures (containing relatively heavy calculations) were written in C/C++ and were lifted to some of the most popular scripting languages (PHP, Perl and Phyton). The lifting process passed smoothly and the results were encouraging – using of lifted procedures led to significant decreasing of execution time, compared with the time necessary for same calculations but written in the corresponding scripting language.

The second experiment was dedicated to execution of call (local, not remote) of procedure written in C from procedure written in scripting language. One of the standard interoperability protocols, created with RPC, was extended to a front-agent and lifted to Python-level. Then a procedure written in Python called successfully a procedure written in C. Some experiments for remote call of procedure written in C from procedures written in scripting language are in progress. It is clear, that more efforts will be necessary for implementing of the end-agent.

As a result of experiments we could make the following conclusions:

- Proposed in the paper approach is **quite realistic** and could be used **for achieving cross-technological interoperability** of the applications in a web-based software solution;
- The proposed approach is **very promising in sense of time consuming** and could be much more appropriate than some other approaches – for example, using XML as an interoperability mediator;
- The proposed approach **could be implemented** with minor extensions of the existing tools for creation of interoperability protocols and lifting of procedures. RPC and SWIG are very good base for start of the implementation;
- Both discussed mechanisms are **not easy** and their usage will be a true challenge for some developers. That is why a corresponding interface (shell) for ordinary users has to be provided too.

Bibliography

[Maneva, Manev, 2008] N. Maneva, Kr. Manev. On the models of development and distribution of Software, Interntional Journal of Information Theories and Applications, No. 15, 2008.

[OMG, 2008] Model Driven Architecture, http://www.omg.org/mda

[Ousterhout, 1998] Scripting: Higher Level Programming for the 21st Century. IEEEComputer, March 1998.

[Marshal, 1999] D. Marshal. A tutorial on ONC RPC, http://www.sc.cf.ac.uk/Dave/C/node33.html, May 1999.

[Java RMI, 2007], The Java Tutorials. RMI. http://java.sun.com/docs/books/tutorial/rml, 2007.

[RPyC, 2007] Remote Phyton Call (RPyC). http://rpyc.wikispaces.com, 2007.

[SWIG, 2007] Welcome to SWIG. http://www.swig.org, 2007.

[SWIG, 2002] SWIG :Pointers to functions and callbacks, http://www.garyfeng.com/wordpress/2002/11/27/swig-pointers-to-functions-and-callbacks/, 2002.

Authors' Information

Stefan Palanchov *– Manager; STEMA-SOFT, St. Ivan Rilski, No. 27, vh. A, Varna-9009, Bulgaria; e-mail: s.palanchov@stemasoft.com*

Alexander Simeonov *– Manager; Inovative Web Solutions, Mladost 4, 448, vh. 3, Sofia-1715, Bulgaria; e-mail: simeonov@stemasoft.com*

Krassimir Manev *– Associated Professor, Faculty of Mathematics and Informatics, Sofia University, 5 J. Bourchier str, Sofia-1164, Bulgaria; e-mail: manev@fmi.uni-sofia.bg*