TRANSPORTING OBJECT ORIENTED DATA BETWEEN DISTRIBUTED APPLICATIONS

Adrian Pelcz, Francisc Sisak, Liviu Perniu

Automatics Department, "Transilvania" University of Brasov, M.Viteazu Street, no.5, 500174, Brasov, Romania, phone/fax: +40 0268 418836,
apelcz@vision-systems.ro, sisak@unitbv.ro, liviup@marconi.unitbv.ro

Usually, the components of the distributed software applications are developed using the same technology, but sometimes, the components must be developed using different technologies. This is the case where data communication becomes a difficult task because even though these technologies are compatible at network protocol level, data is transported as byte arrays. Modern software development languages provide object-oriented support, and thus, data is organized in objects. Object serialization could be a difficult task for programmers, as they must write mirror classes, containing functions for packing and unpacking data as byte arrays. A minor change in a class structure or writing a new class could take large amounts of time. Our method offers a lightweight solution for data transportation between these applications using TCP/IP or HTTP, as well as a transport class generator for C++ and Java. The framework reduces development time for the data transportation code to minutes.

Keywords: software, development, distributed, object oriented, data transport

1. INTRODUCTION

The tools or technologies required to write the code for developing distributed applications are not fully compatible, or the compatibility level is low. Such an example is with C++ and Java environments. The basic principles of object-orientation apply to both languages and the syntax is alike. The technologies based on these languages offer compatibility at TCP/IP or HTTP level for communications between applications. The high level of today's programming environments does not fit the TCP/IP paradigm. Evolved languages offer support for data serialization over all kinds of data streams (files, memory, network). Both Visual C++ and Java offer support for object serialization. When we talk about developing a distributed application using Java for instance, passing objects or arrays of objects, over the network, between the modules of the application does not represent a problem at all. Any object can be sent very easy from one module to another over the network. The same discussion applies to C++ applications.

If the project requirements say that one module should use Java and other module should use C++ as technologies, the main concern for programmers is how to make the communication between the modules more efficient.

In this presentation, we will use typical client-server architecture, where the client is developed using C++ and the server is developed using Java.

If the application is very small, the problem seems simple: the programmer(s) establish a particular communication protocol that fits their application and develop

code for that. The protocol is reduced to transforming any kind of data to be sent into byte arrays, with a non-standard form. This approach starts from the idea that the data sent is exactly what is needed for each operation. Very often this solution is rigid and brings the following problems:

- No code reusability;
- Slow development;
- Complications when the applications grows larger;
- To make changes into a functionality (adding an extra information to be sent) implies redesigning the telegram for that function and changing the corresponding parts of the code;
- Not object oriented (even though behind the byte arrays we find objects in C++ and Java, this "customized" serialization for the data, dims the class-object correspondence between the two modules).
- No uniformity in the client-server communication zone;

This problem has many solutions from recommendations and standards, to enterprise solutions like CORBA (Common Object Request Broker Architecture) [5]. Microsoft COM/DCOM or Sun RMI, are designed to be used into distributed applications, which are implemented using the same technology for all the components, and thus, these technologies do not work into heterogeneous applications where the components use different technologies.

CORBA allows data transport between C++ and Java relatively easy, once the programmers get to know it. The principle is to create .idl files (using Interface Descriptor Language). These files are used by a class/file generator, which creates C++ and Java skeleton files, based on the IDL definition.

Many of the existing solutions imply installing third party software applications or server, technical expertise in using, configuring and installing them, and possibly extra costs for the customers of the developed software, due to licensed third party solutions.

We propose a simple, lightweight solution as a development framework for passing objects between C++ and Java distributed applications, targeted to software developers who do not want to use heavy, third party solutions for making an efficient development.

The proposed methodology is not related to the communication protocol at all, as it is just a way to pack and unpack objects between the two programming environments. Thus, it does not matter if an application uses TCP/IP or HTTP for communication.

2. SERIALIZATION AND DATA TRANSPORT

In order to create an efficient framework for developing C++/Java mixed applications we have to provide two main features:

1. A unified manner for transmitting serialized objects (or arrays of objects) easy between C++ and Java.
2. A tool for serializing objects easy.

We have to create a mechanism which is able to send and receive language specific data, but also compatible with the other language. For an object to be serializable, we must provide a function which returns a byte array that contains all the information stored in that object and also a function that can recreate an object from a given byte array.

When we have the object serialized, it is easy to send it over the network. Here we encounter the other side of the problem. The language is different and so are the objects. Therefore, here we must create a twin class, with the same properties as the original class, and the same functions for serialization and de-serialization. Practically, we must write the same class in C++ and in Java and must provide functions for serialization and de-serialization for each class (totally, 4 functions). When the class structure consists of many member variables, with different data types, writing a serialization function becomes difficult and the development time increases dramatically with the members number. In addition, making a change in a structure class implies changing all the four functions (this takes a lot of time).

The idea is to create a set of functions that organize serialized objects into transmittable telegrams, and free de programmers from the task of keeping in mind the data transport problem. Ideally, programmer should just define the properties for the classes and the data processing functions, not the data transport functions and classes. This allows the programmer to focus on the business-logic of the application and not on how data is transported. Along considerably shorter time for development, the possibility to introduce bugs is decreased also.

For achieving this goal, we must automate the development for transport classes (create a program that generates the C++ and Java classes, based on an enumeration of member variables). In addition, we must provide classes for C++ and Java that handle the data transport, making it transparent to the programmers. Like this, working with C++/Java would be just as simple as working with the same development environment/ language.

3. THE TRANSPORT CLASS GENERATOR

For addressing the problem of creating compatible classes for C++ and Java, we have developed a class generator tool. For developing this tool, we used the Java language.

A transport class usually is consisted of a set of private member variables, and access methods/functions (get and set methods). Such a class has the role of retaining a certain data structure for transporting it between functions, modules, applications, etc.

3.1 File format and supported data-types

The program runs as a command line application and takes the input file name as paramenter.

The input file contains the member variables of the classes enumerated as <data-type> <variable name>, on each row. The data-type in the input file is

specified as Java primary data-type. The program will determine the C++ equivalents for the specified data-types.

The current version of the class generator is able to create classes that contain only the following Java data-types: String, int, long, boolean, float and Timestamp.

The class generator will parse the input filename and determine the variables to be put into the generated classes.

3.2 Generating the code

The class generator must create three files. The .java file, containing the Java class definition, the .h file containing the C++ class declaration and the .cpp file, containing the C++ class function members definition.

The generator parses the input file line by line and for each member found, it creates the specific code.

All the three output files are generated at the same time, passing through the class definition only once. There are also some special elements included in the transport files, for allowing easier integration into the working project, which will be detailed further.

The most complicated and important part of the code generation is the serialization. The program must generate code that can serialize data into byte arrays and de-serialize from byte arrays.

The program takes the variable members in the order into which they appear into the input file and transforms each into a byte array. The individual byte arrays are concatenated and the resulting byte array is the serialized object. On the other hand, the de-serialization function, can parse a given byte array and transform it into an object. The condition for this to happen is that the information to be stored in the exact same order and the processing should be the exact reverse.

For avoiding numerical representations on different machines/operating systems, we transform the numerical values into bytes using subsequent divisions by 256. Like this, we have the number as bytes sequence, starting from the most significant byte to the least significant one. When recreating the numerical value, we use a reverse process, and like this, the number representations (such as little endian, big endian do not matter).

Given this structure, when the de-serialization function receives a byte array the process for transforming it into an object is absolutely determinist, because the order of variable members is the same as for serialization, as both functions are generated at the same time, and on the same basis.

4. FRAMEWORK DESCRIPTION

For the management of the data communication and function calling from a C++ client on a Java server, we have developed a set of classes and a work procedure that helps the developers to easy add new function calls and/or make changes to already developed classes or functions.

Following, the Java and C++ parts of the framework are described.

4.1 The Server Side (Java application)

We look at the server as a request solver. Usually, a request has an identifier, which separates a request from another. The server reads the data from the C++ client as byte arrays. The data received is being split into a command ID and a command body (the parameters for the function).

The server instantiates a class that executes the specific function, corresponding to the received command ID. This class calls a function passing to it the received byte array [2]. For easy management of parameters, we use the transport class generator to create classes that contain properties. These functions will execute the needed operations and will return a byte array, which will be sent back to the client as response. This byte array is also generated as serialized object (instances of transport classes created with the transport class generator).

For uniformity, we use the Java polymorphism [1]. This means that all the objects that support serialization must implement an interface. This interface is called `ITransport`. `ITransport` defines two functions: `GetTelegramLength` and `GetAsByteArray`. The first function returns the length of the byte array that would be the serialized object, and the second function returns the byte array containing the serialized object. Like this, due to polymorphism, we can define functions that work with `ITransport` objects disregarding the actual object definition. In addition to implementing this interface, any object that is generated using the transport class generator will have a constructor that takes a byte array as parameter. The constructor does the de-serialization and the `GetAsByteArray` function does the serialization.

Because the functions from the transport class provide support for serializing a single object, and because in many cases we need to send lists of objects (a list of users for instance), we have developed a special class called `TransportParser`. This class has two roles:

- To pack an array of objects that implement `ITransport` into a byte array (using the serialization function for each object);
- To return from a given byte array the number of elements and to return a given element as byte array (the actual object is constructed calling the constructor with the returned byte array as parameter).

4.2 The Client Side (C++ application)

On the client side, we have classes very similar to those on the server. There is virtual class (`CTransport`), from which any object created by the transport class generator is derived. The `GetTelegramLength` and `GetAsByteArray` are the same as those from the server from a functional point of view [6].

There is also a `TransportParser` class, which allows creating of a byte array from a single object, compatible with the byte array generated by the Java `TransportParser`. For the correct mapping of the server functions, the same constants (as name and value) are defined both on the server side and on client side.

5. ADVANTAGES PROVIDED BY THE FRAMEWORK

The most important advantage provided is a considerably shorter development time for data transport due to the automation of code writing for transport classes and data transport procedures.

In addition, the data transport between modules becomes very transparent to the developers because they do not have to concern anymore about packing and sending data - they just call the same functions and work the same way, just change the class name and add a function ID, this giving uniformity to the code.

A high degree of flexibility is obtained because a change in a transport class can be done in a few minutes and the transport code needs no changes. Just the .tcc file needs to be changed, the classes regenerated and overwritten.

Better focusing on the business-logic by programmers is obtained because the data transport between modules is almost as simple as a function call. Actually, the framework resembles a remote function calling mechanism.

6. FUTURE DEVELOPMENT

We intend to improve more this system in the following directions:

- Add new supported data types to the existing ones;
- Improve the transport class generator by adding more options for class generation;
- Add support for recursive serialization (allow the class members to be instances of classes generated with the transport class generator);
- Provide a graphical user interface for the transport class generator;
- Create a true remote function calling mechanism, based on the existing framework.

7. CONCLUSIONS

The presented framework has already proved it's utility in the development of software projects where it reduced the overall duration for code writing with about 25%.

The possibilities to extend the framework are promising and it could become a very powerful tool in the development of small-to-medium distributed applications.

8. REFERENCES

[1] Horstmann C.S., G. Cornell, *Core Java 1*, *Core Java 2*, The Sun Microsystems Press.
[2] Harold E.R., *Java network programming - Second Edition*, O'Reilly, 2000.
[3] http://java.sun.com/
[4] Parks T.M., D. Roberts, *Distributed Process Networks in Java*, International Workshop on Java for Parallel and Distributed Computing, Nice, France, 2003.
[5] Killijian M.O., J.C. Ruiz, J.C. Fabre, *Portable Serialization of CORBA Objects: a Reflective Approach*, OOPSLA 2002, Washington - USA.
[6] Microsoft Studio Developer Network for Visual C++ .NET (electronic documentation available from Microsoft)