RC23581 (W0504-036) April 8, 2005
Computer Science

IBM Research Report

A Method for Enabling Cross Platform Function Calls

Leonard Berman, David Jensen
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A method for enabling cross platform function calls

Leonard Berman*and David Jensen†

April 6, 2005

Abstract

We describe a new method for enabling cross platform function calls. Using our methodology the cost of providing `M` libraries to `N` runtimes is $\mathcal{O}(\mathtt{N}+\mathtt{M})$ rather than $\mathcal{O}(\mathtt{N}*\mathtt{M})$. The method relies on a portable, introspective data structure to transport invocation requests and an IDL which provides enough information to allow automated data marshalling between the portable format and the runtimes of the client and server.

The method, which is called *ApplicationWeb,* requires one infrastructure component per runtime and one artifact per library exposed. It permits exposure of functions with complex signatures (including output arguments), and requires writing no library specific code.

Our implementation has been used to provide function call access to libraries, such as LAPACK[1] or COIN[2], between runtimes including *C, Java™, C++, FORTRAN, R*[3]*, perl, Visual Basic®, zLinux*[4]*, *nix,* and *Windows®*.

1 Introduction

The importance of connecting components from different runtimes and platforms is well known. Different languages are effective for different sorts of applications and most vendors agree that, to provide the best solutions to complex problems, programmers must be able to combine codes independent of their implementation language, without excessive effort.

In the past, the scope of this cross platform function call problem was generally considered to be a single machine; however, today as loosely coupled systems of heterogeneous machines and service oriented architectures become more widespread, this limitation is no longer appropriate. Today, a programmer needs transparent access to libraries and services independent the machine providing the functionality.

*IBM Research Division, PO Box 218, Yorktown Heights, NY 10598. *e-mail: namreb@us.ibm.com*
†IBM Research Division, PO Box 218, Yorktown Heights, NY 10598. *e-mail: davjen@us.ibm.com*
[1]http://www.netlib.org/lapack/
[2]http://www.coin-or.org/
[3]http://www.r-project.org/
[4]http://www-1.ibm.com/servers/eserver/zseries/os/linux/library/rpapers.html

The goal of our work is to minimize the increase in effort, which is required to utilize or to provide library access, due to the remote nature of the interaction. Put another way, the question of whether a given library is provided locally or as a remote service should be irrelevant from the standpoint of programmer effort.

We will say that an infrastructure which achieves the minimum, i.e. provides remote library increase with no increase in programmer effort, is providing *automatic cross platform function calls.* We note that in such a system, a programmer can not be required to write any data marshalling code.

1.1 Cross Platform Function calls

Providing cross platform function calls is a complex problem with a long history. Stevens [1] discusses the steps of the problem as well as three early systems for its solution. More recent efforts include: CORBA[5], SWIG[6], JAVA/JNI,... Systems such as Microsoft's DCOM® and Apache AXIS[7] address the related problem of performing remote function calls when both endpoints provide the same runtime. Each of these efforts provides a different procedure for one or more of the steps involved in the complete problem.

We will begin by presenting our perception of the fundamental problem. Two tasks must be accomplished in every function call.

1. Data and service requests, represented as byte sequences, must be transported between the calling and called environments.
2. These byte sequences must be interpreted as semantic objects and processed.

These tasks exist whether the two routines are procedures defined in a single file and running in a single process or are processes residing on different machines.

In a typical uniprocessor environment, the common runtime (shared by caller and called) provides machinery for both of these tasks. Data is transported (and results returned) *via* parameter passing or global addressability inherent in the common runtime, encoding issues (endianess and character encoding) are non-existent, and data interpretation is specified using a combination of a type system, e.g. C header files, and an informal agreement[8] between calling and called procedure. Depending on the type system, the amount of information contained in the "informal agreement" may vary.

In the more complex cross platform environment, information contained in an "informal agreement" is of no use. The main contribution of our work is to describe

- an extended type system which can specify method signatures completely for a large subset of `C` and `FORTRAN` functions, and
- an implementation which uses this type system to provide automatic cross platform function calls for these functions.

[5]http://www.corba.org
[6]http://www.swig.org/
[7]ws.apache.org/axis/
[8]The prototypical agreement is that the shape of a returned array is specified by another returned value.

The systems mentioned above begin with the `C` type system. This type system is adequate to describe the signatures of functions whose parameters and return values do not involve array formation or indirection. As a result, the cross runtime systems mentioned above can (or could be modified to) provide automatic remote function calls for functions with arguments and return values of these basic types. (We note that AXIS can provide automatic function calling involving bean types when both endpoints are Java; however, this relies on the Java runtime, and while a very useful facility, it does not address the problem of providing cross platform support.)

As is well known, the `C` type system can not fully describe data which involves indirection or arrays. In order to support function calls for methods whose signatures involve these complex types, the artifacts associated with the previous systems (i.e. the specification file or generated wrappers) must be modified, in a runtime specific manner, to reflect the semantics of each exported function. To put this more plainly, code must be written to describe the details of the data marshalling.

The ApplicationWeb methodology uses an extended type system, which includes a `String` type and the array formation operator. It can express array shape constraints among function parameters, and the ApplicationWeb infrastructure can use this information to provide automatic remote procedure calls for most C functions, including those with arguments which involve indirection, arrays, and null terminated character arrays in addition to the basic built-in and structure types mentioned above.

1.1.1 Example

A simple example will illustrate the distinction. Consider a C function with the following signature and behavior:

- **int** fnArgIntArr(**int**∗ arrayIn, **int** len , **int**∗∗ arrayOut);

Where `len` is the length of the result of `arrayIn`, and where on return,

- the elements of `arrayIn` will be sorted,
- the return value will be the length of `arrayOut[0]`, and
- `arrayOut[0]` will hold the non-zero elements of `arrayIn`.

ApplicationWeb uses an XML based IDL which can express the shape of both `arrayIn` and `arrayOut`. Given this specification file, an ApplicationWeb server is started at a known ip address, and the shared library implementing the function and the IDL file is placed in a known location relative to the server. Once this has been done, for a supported client runtime, ApplicationWeb supplies infrastructure that enables a function call using the generic code shown in figure 1.

The key point of this example is that a single, runtime independent specification file allows calls to be made from a client in any supported runtime to a server providing the described functionality in any other supported runtime. Data marshalling for the arrays is handled by the infrastructure given only the specification file.

This is in contrast to systems like CORBA or SWIG where either the specification file or generated wrapper code would require modification depending on the endpoint runtimes.

```
awObject = new ApplicationWebObject(...libraryXML...);
awObject . configureTransport(...);
awObject . setMethodName(methodName);
for( I = 0 ; I < k ; I++ ){
    awObject . addArgument(argumentName[I], inValue[I]);
}
awObject = awObject.call();
for( I = 0 ; I < k ; I++ ){
    outValue{I] = awObject . getArgumentValue(argumentName);
}
returnValue = awObject.getReturnValue();
```

Figure 1: Example client code

1.2 Advantages and Disadvantages

Nothing is perfect. Advantages(+) and disadvantages(-) include:

+ Reduced opportunity for error. Since the required interface specification is supplied only once in a runtime independent manner, there are fewer opportunities to introduce errors and inconsistencies than there would be if custom data marshalling code were required for each client runtime.

+ Division of labor. The specification of the interface semantics can be furnished by the interface provider who may not be knowledgeable about the runtimes from which access will occur, while the runtime specific infrastructure can be provided by the platform provider.

  For example, the specification file for LAPACK was produced by a perl script which extracted the required information from LAPACK documentation. The adapter which connects the FORTRAN and C runtimes was produced entirely independently.

+ No modification of generated artifacts. Previous systems required modification of generated artifacts in order to perform data marshalling.

- Startup costs. The complexity of the platform specific infrastructure component supporting a runtime is greater than that required when extending a runtime by a single library.

- Efficiency. It is not possible to supply custom code for data marshalling.

2 ApplicationWeb

In this section we will discuss a number of issues which we have addressed in ApplicationWeb.

To recapitulate, the ApplicationWeb IDL can express both constraints between function arguments and information concerning storage ownership which the Appli-

cationWeb infrastructure utilizes to automate data transport and server side storage management for commonly occurring function types.

ApplicationWeb is built from

- A platform independent, introspective representation for data transport,
- An XML based IDL, which provides the information required for data marshalling, storage management and function calling, and
- An infrastructure that uses the information provided by the IDL to perform data marshalling and function invocation in each runtime.

In our implementation, the data representation is a single recursive C structure; however, any introspective representation could be substituted. The function call process decomposes into the following tasks.

1. Marshalling data between the transportable format and the runtimes of client and server.
2. Transporting data between client and server,
3. Processing the client service request, i.e. invoking the function.

Once the transportable object has been produced, the actual transport is well understood so the remainder of this section will discuss the two remaining problems: how to map data to/from the transportable format, and how to process the client service request.

2.1 Data marshalling and the IDL

We call the XML file describing a library the *augmented specification file*[9]. Conceptually, the augmented specification file contains the knowledge that a programmer familiar with the library utilizes in order to use the library effectively. That is, it is a machine readable representation of the excellent manual which comes with all libraries. It allows us to map data between the transportable format and the representation required by the library runtime.

The next few paragraphs touch on some of the key features of our IDL.

Basic types The ApplicationWeb IDL currently supports the following fundamental data types: byte, short, int, long int, float, complex, double, long double, and a String (null terminated sequence of bytes) type as well as structure and array formation. ApplicationWeb does not currently distinguish between signed and unsigned integer types.

Array shape In order to support data marshalling for array arguments, the IDL allows array dimensions to be specified by MathML[10] expressions involving constants, the value of arguments/fields, or the value returned by library functions applied to arguments of the current function. The IDL also allows the specification of additional

[9] Augmented because it contains more information than would be contained in a C header file.
[10] www.w3.org/Math/

constraints among method arguments. These constraints are also expressed as MathML expressions involving arguments and structure fields.

Resources Library functions may allocate resources which need to be managed. For example, depending on the implementation, the resource associated with argument `arrayOut[0]` of `fnArgIntArr` (section 1.1.1) might need to be released. The IDL allows an arbitrary function to be associated with each argument and return value. This associated function is called with the given value after function invocation and output data marshalling. It is typically used for freeing storage.

Tokens In the case of basic types, it is possible to transport actual data between client and server; however, this is not always desirable since

- a large, complicated data structure is frequently of little use outside of the runtime that constructed it, and
- transporting data means discarding information about referential equality of server side objects.

Other types such as `java.lang.BigInteger` or COIN's `BCP_MemPool`, may not have client side representations although these server side data types can be created and manipulated using methods whose arguments eventually resolve to the basic types.

To address these data transport issues, our infrastructure can create, maintain, and transport tokens which represent server data and allow the client to manipulate these values. These tokens allow data structures created in the server runtime to be maintained across calls and to support libraries whose functions depend on equality of reference (as opposed to value) arguments.

2.2 Infrastructure

The infrastructure utilizes the information contained in the augmented specification file to map data and to invoke methods on the server. Each of these processes can be performed in two ways: interpretively or by generating and compiling code. Each approach has advantages and disadvantages which we will discuss.

2.2.1 Interpretation.

Interpretation requires having the IDL file available at the use site; however, it eliminates the need for compilation or any pre-deployment processing of the augmented specification file. It requires one artifact, the augmented specification file, independent of the number of runtimes which expose or utilize a given library.

Interpretation is likely to be less efficient than generating and compiling code; however, for computationally intensive operations, this overhead is insignificant.

For C and FORTRAN servers, our infrastructure performs function calls interpretatively by including an assortment of function prototypes and casting the library function pointers appropriately. This approach limits the variety of function signatures and library compilation flags which ApplicationWeb can support in a fixed build.

Analogous problems arise in marshalling data to structures. We have not found these limitations to be serious problems since a small number of prototypes can accommodate a very wide assortment of library functions and most structures can be mapped easily. These problems do not exist in runtimes, such as Java, which support reflection.

2.2.2 Code generation.

Code generation removes the need for the IDL file at use sites; however, compiled helper libraries are required instead. Since the helper library is runtime specific, this approach requires the preparation of one artifact per library per runtime which complicates deployment.

Since the helper library can include prototypes for any library function and data types, this approach supports functions with arbitrary signatures and libraries that have been compiled with arbitrary flags. In addition, data marshalling and function calling through compiled artifacts is likely to be more efficient than interpretation.

SWIG and CORBA use code generation. An augmented IDL of the sort described here could be incorporated into these systems to enable them to generate wrappers which transport array objects without hand modification.

2.2.3 Client side infrastructure

The example pseudo-code shown in figure 1 illustrates, the sort of client code which a user might write to use ApplicationWeb. As noted, this is essentially language independent; however, it is cumbersome and distracts from the application logic of the client code.

In order to improve usability, for each supported client runtime, we provide an XSLT[11] which generates and encapsulates the wrapper code. Using our Eclipse plug-in with the prototype shown in section 1.1.1, we generate a Java class with the following public method:

public int fnArgIntArr(**int**[] arrayIn, **int** len, **int** [][] arrayOut);

The behavior of the Java method is indistinguishable from that of the original `C` method, i.e. if

- On input: arrayIn={1,0,2} and arrayOut=new int[][]{null}; then
- The function returns 2, and on return, arrayIn={0,1,2} and and arrayOut[0]=={1,2};

Once these additional client artifacts have been generated, the IDL file is no longer required at the client.

2.3 Performance

The goal of ApplicationWeb is to simplify development of applications which combine complex components independent of their runtime requirements. Performance is a concern; however, it is not our primary concern. Applications of interest, such Monte

[11] http://www.w3.org/TR/xslt

Carlo simulation or large scale optimization, have significant computational requirements which grow rapidly with data size. For these sorts of applications, the computational requirements overwhelm data marshalling costs for modest input sizes. We would not suggest that our methods be used to replace custom methods in performance critical applications with low latency such as the eclipse SWT implementation, but for computationally demanding tasks, we feel the performance is more than adequate.

Our platform independent representation transports data as a tree with one node per parameter, structure field, and array. The primitive elements themselves are transported in binary format. We have been careful to use algorithms linear in the size of the data which must be transported and the result is a constant latency overhead per field in addition to the costs for data movement (and byte reordering if required).

3 Summary

We have described a methodology that combines a runtime specific infrastructure and a single runtime independent artifact (the augmented specification file) to provide a generic function invocation service. The service allows programmers to access the functionality exposed by a library uniformly, irrespective of the environment of either the client or the server machine, with no more effort than would be required to access the library locally. While the library provider must furnish the augmented specification file, we suggest that this is the equivalent of providing a thorough manual which is part of a library programmer's job.

Once the augmented specification file has been constructed, the methodology does not require any further input (beyond invoking the call).

We also described an implementation of the methodology which uses an interpretive infrastructure to realize the methodology, and observed that the idea of an augmented IDL could be incorporated into systems such as SWIG or CORBA to enable them to generate wrapper code which did not require hand modification.

References

[1] W. R. Stevens, *UNIX Network Programming.* Prentice Hall Software Series, Prentice Hall, 1990.

3.1 Acknowledgements

We would like to thank Vivek Sarkar for reading and commenting on an earlier version of this paper.