MARSHALLING IN DISTRIBUTED SYSTEMS: TWO APPROACHES

K.V. Dyshlevoi, V.E. Kamensky, L.B. Solovskaya

In distributed system different modules can use different representations for the same data. To exchange such data between modules, it is necessary to reformat the data. This operation (called *marshalling*) needs some computer time and sometimes it is most expensive part in network communication. The article discusses some problems and concrete implementation examples of marshaling procedures.

1. INTRODUCTION

In general, modern computer systems are distributed systems. Such systems are implemented as computer networks, whose components jointly use and exchange data. To describe distributed systems at abstract level client-server model can be used. In terms of this model, client makes invocation, containing all necessary parameters, on server. Server receives the invocation, performs needed actions and returns the results back. Parameters and results can be exchanged between different platforms, which use their own data representation. Platform means programming language and compiler too, rather than hardware and operation system only.

Client and server can interact directly. In this case, they have to manage request passing themselves, by their own protocol. To simplify and unify client-server interaction some mediator-systems, such as ONC RPC, DCE PRC, OMG CORBA, can be used.

Mediator-systems use some specific communication protocols and encoding rules for data passing. There are some data coding standards, for example, CDR, BER, NDR, XDR, and appropriate communication protocol, GIOP, OSI RPC, ONC RPC, DCE RPC.

In this way, mediator performing client-server interaction, have to be able to reformat data. The operation of data presentation conversion for further network passing is known as *marshalling*. They can say about marshaling for passing data in network without restriction of the task. Just notice, that the task of heterogeneous data conversion into a bit stream has appeared earlier and has it's own meaning.

So, “overhead expenses” for data marshalling are inevitable in distributed systems. Marshalling can be the most expensive part in network communication, particular, in LAN, when the time needed for network passing itself is less than time needed for marshalling.

There are various approaches to increase the efficiency of marshalling. One possible approach is an introduction of new encoding rules in order to reduce size of data passed in network and time needed for their marshalling [3]. Another approach is to realize fast encoding/decoding procedures by means of a hardware encoder/decoder or by optimizing implementation of a software encoder/decoder.

Two implementation techniques are commonly used for the marshalling routines - *interpreted* code and *compiled* code, which can be extended by *inline*-substitutions. These techniques have a well-known trade-off between code size and execution speed. Interpreted code is compact, but slow, compiled code is fast, but memory-consuming.

To illustrate possibilities of interpreted and compiled techniques in sense of marshalling implementation, a model language L is considered. It consists of two basic types T1 and T2 and one complex type *struct,* which resembles C type *struct*. User defined *struct* types are recursive and can be of any depth. There are examples of acceptable types - *struct S1 {T1 f1; T2 f2}* and *struct S2{S1 d1; T1 d2}*. Special type *TypeInfo* describes internal structure of types, coding subsequently their elements. Type T1 is coded by 1, type T2 - by 2. The code of user defined types begins with 3, finishes with 0 and contains codes of their elements. For example, code for type S1, declared above, is “3120” and code for type S2 is “3312010”.

The simplest example of marshalling interpreter, which supports all types of language L, is following:

```
typedef char * TypeInfo;
void interpr (TypeInfo &t, void *data) {
    while (*t) {
        switch (*t) {
        case’1’:
            put_T1(data);
            ((T1 *)data) ++;
            t++;
            break;
        case ‘2’:
            put_T2(data);
            ((T2 *)data) ++;
            t++;
            break;
        case ‘3’:
            interpr(++t, data);
            break;
        case ‘0’:
            return;
        }
    }
}
```

To use compiled code for marshalling, it is necessary to generate individual marshalling procedures for each user defined type. For example, marshalling procedures for types S1 and S2 are the following:

```
put_S1 (S1 *data) {
    put_T1 (&(data->f1));
    put_T2 (&(data->f2));
}

put_S2 (S2 *data) {
    put_S1 (&(data->d1));
    put_T1 (&(data->d2));
}
```

The article introduces some concrete implementation examples of both methods and comparison of their abilities on some benchmarks. Marshaling procedures is considered in the context of Object Request Broker (ORB).

The rest of this paper is structed as follows: Section 2 proposes the notion of marshalling in more details. Basic concepts of ORB technology are described in Section 3. Section 4 and 5 give concrete examples of interpreted and compiled code for marshalling, respectively. Results of experiments are discussed in Section 6. Section 7 gives our conclusions and possible areas for future research.

2. MARSHALLING

Marshalling is a data presentation conversion, performed according to special rules, usually for network transfer. The following data presentation factors have to be took into account to perform marshalling.

Different platforms use their own character formats (ASCII or EBCDIC), integers and floats formats (IEEE, VAX, Cray, IBM). For example, the scheme is acceptable - the native numeric representation is two's complement for integers, ANSI/IEEE single/double for floats and also that characters are in ISO Latin/1 (an ASCII superset addressing Western European characters).

Byte order is basically dependent on processor (*Big Endian* or *Little Endian*). But some processors (MIPS, UltraSPARC, PowerPC) support different byte orders as required for different software environments.

There are different alignment strategies. “Natural alignment” is a policy that the processor controls the alignment of data based on its type. It means that the data of basic types are located in memory from the boundaries divisible by their size. Some CPUs have a maximum alignment requirement of two or four bytes, others have some type-specific exceptions to the normal "alignment ==

size" rule. “Fixed alignment” ignores data type when establishing alignment; not all processors support such policies.

Also, realignment is required for the components of record or structure types, since different CPU’s use different rules for positioning structures fields in memory.

Linearization is required for data structures that are stored in non-contiguous memory sections, such as dynamically allocated tree structures.

So, marshalling procedures have to have information on data format for current platform, convert it into some standard format used for network transfer, get data from network and be able to decode them back from standard network format into this platform one. In this way, marshalling includes two mutually reverse procedures - encode and decode.They have symmetrical algorithms, whose differ just by basic primitives ‘put’ into and ‘get’ from some buffer. But decode procedures are always more complex due to the requirement for error-checking and some memory allocation. Further we shall call those procedures marshalling and unmarshalling, respectively.

3. BASIC NOTIONS
3. 1. OBJECT REQUEST BROKER

Marshalling procedures are discussed in the context of Object Request Broker (ORB) design. Also, we shall consider only object-oriented distributed systems.

Some mediator is needed to allow a client to send a request to an object implementation transparently. The transparency means independence of platform (hardware, operating system, programming language) where the object is resided. ORB plays the role of such mediator and provides interoperability between applications on different machines in distributed environments.

Main functions of ORB are:

- to find the object implementation for the request;
- to pack the request and it’s parameters, pass this message to object implementation;
- to wait for the results, decode and return them back to client.

Additionally, ORB supports security, authorization, persistency and some others services needed in distributed environments.

Usually, two ORBs take part in the interaction - client-side and server-side broker. Client’s broker passes the request from client object to server and received results. Vise verse, server-side ORB gets the request, passes it to the object implementation and returns results. In general case, an object can be a client and a server at the same time. So, client and server brokers are ORBs with full functionality.

The ORB we are going to discuss and objects to be handled by this broker are implemented in C++ language. The ORB supports OMG CORBA standard.

3. 2. OMG CORBA 2.0 STANDARD: IDL, GIOP, CDR

CORBA is a standard of ORB technology produced by OMG (Object Management Group). OMG is aimed at design of open standards that provides interoperability between object-oriented applications in distributed environment. CORBA is a most popular development of this consortium. The standard is supported by a lot of companies, e.g. DEC, HP, SunSoft Inc.

CORBA is based on the OMG object model. It is classical client-server model that allow objects to interact by means of sending and receiving messages. It is important to notice, that interfaces are separated from implementations and object model defines interfaces only by means of IDL (Interface Definition Language).

IDL serves to describe the interfaces that clients call and object implementations provide in the particular programming languages independent manner. An interface description consists of a set of named operations and the parameters to those operations. Syntax of IDL is mapped onto languages, that are used to implement client and server objects (these languages may not coincide). Syntax of IDL resembles C++ language. IDL is just a descriptive language, which does not contain operators.

IDL is a typed language. All types, available by IDL syntax, can be divided into two groups - basic and constructed types. Basic types include signed and unsigned integer types, floating-point types (IEEE single-precision and double-precision floating point numbers), char type (represents the ISO Latin-1 character set), boolean and some others types. *Octet* is a special 8-bit type that is guaranteed not to undergo any conversion when transmitted by the communication system. *Interface* is the main IDL type needed for object descriptions. Interface body can contain operation, type, constant and some other declarations. Constructed types involve *struct*, *union*, *array* and *sequence* types. *Struct* type is analogous of C struct type. *Sequence* and *array* types are the sequences of elements of one type variable and fixed length, respectively. Semantic of *union* type resembles C union type expanded by case descriptors.

CORBA describes protocol-level ORB interoperability too. GIOP (General Inter-ORB Protocol) defines protocol of ORB interconnection, which can be mapped onto any connection-oriented transport protocol. For example, there is specific mapping of the GIOP which runs directly over TCP/IP connection, called the IIOP (Internet Inter-ORB Protocol). GIOP declares some additional requirements needed to pass messages between ORBs. GIOP consists of elements:

- CDR (Common Data Representation);
- GIOP message format;
- GIOP transport assumptions.

CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation used in the GIOP messages. CDR concerns the following features:

- each GIOP message contains a flag that indicates the appropriate byte order;
- basic IDL data types are aligned on their natural boundaries within GIOP messages;
- CDR describes representations for all IDL data types.

CDR specification is based on the notion of *octet stream*. *Octet stream* is an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine.

4. MARSHALLING INTERPRETER
4. 1. TYPRCODE

Marshalling of the basic types is performed by means of set of basic primitives. Also those primitives are used by marshalling procedures of complex types. This implementation assumes that the native numeric representation are:

- two's complement for integers;
- ANSI/IEEE single/double for floats;
- ISO Latin/1 characters;
- alignment policy is the same for data of basic types and for the elements of complex types; all compilers we know supports this strategy of alignment.

Also, it supports the five extended OMG-IDL data types, which provide richer arithmetic types (64 bit integers, "quad precision" FP) and UNICODE-based characters and strings. Those types are not standard parts of OMG-IDL at this time.

So, basic primitive implementation uses the standard C/C++ representation for data, and knows how to do things like align and pad according to standard rules. Marshalling procedures for complex types are driven by CDR marshaled representations of TypeCodes. TypeInfo from previous example is a simplest analogue of TypeCode.

There are TypeCodes corresponded to each IDL type, basic and user-defined. It contains all information about internal structure of type. Abstractly, TypeCodes consist of a “kind” field, and a set of parameters appropriate for this kind. CORBA defines *interface TypeCode* and *enum TCKind* types in IDL. Enumeration TCKind is a set of all possible TypeCode kinds, including basic (*tk_long, tk_float*) and user-defined types (*tk_struct, tk_sequence*).

For each user-defined type TypeCode constants are usually generated by IDL compiler according to appropriate IDL specification.

TypeCode implementation is a C++ class:

```
class TypeCode {
public:
// TypeCode kind
TCKind kind(Environment&) const;

// TypeCode equivalence;
// TC_ptr points to TypeCode constants
Boolean equal(TC_ptr, Environment&) const;
...
enum traverse_status {traverse_stop, traverse_continue};
traverse_status traverse(
    const void *value;          // data
    void        *context;       // stream
    ...);
...
TypeCode (TCKind, unsigned char*, ...);
private:
Octet     *_buffer;
TCKind *_kind;
...
}
```

The TypeCode class constructor is initialized by TCKind and char * constant that describes elements of complex types by means of TypeCodes of those elements. For example, to define TypeCode for IDL type *struct Foo {long l; string s;}* it is necessary to point out tk_struct and the following char buffer:

```
const unsigned char tc_Foo[] = {
0, 0, 0, 1,
0, 0, 0, 12, ‘I’, ‘D’, ‘L’, ‘:’, ‘F’, ‘o’, ‘o’, ‘:’, ‘1’, ‘.’, ‘0’, 0,
0, 0, 0, 4, ‘F’, ‘o’, ‘o’, 0,            // type identifier
0, 0, 0, 2,                              // the number of fields
0, 0, 0, 2, ‘l’, 0, 0, 0,                // field l TypeCode (tk_long)
0, 0, 0, 3,
0, 0, 0, 2, ‘s’, 0, 0, 0,                // field s TypeCode (tk_string)
```

```
0, 0, 0, 18,
0, 0, 0, 0
}
```

The method traverse() allow to "visit" each element of a data type in the order those elements are defined in the type's IDL definition.

4. 2. INTERPRETER

CDR, octet stream and accept methods, implements as C++ class:

```
class CDR {
public:
...
// basic marshalling primitives
CORBA2::Boolean put_short(CORBA2::Short s);
inline CORBA2::Boolean put_ushort(CORBA2::Ushort s) {
    return put_short((CORBA2::Short) s); }
...
CORBA2::Boolean get_short(CORBA2::Short &s);
inline CORBA2::Boolean get_ushort(CORBA2::Short &s) {
    return get_short((CORBA2::Short) s); }
...
// marshalling interpreter
static CORBA2::TypeCode::traverse_status encoder(
    CORBA2::TC_ptr tc,          // TypeCode
    const void      *data,      // data
    const CDR       *context,   // octet stream
    ...);

static CORBA2::TypeCode::traverse_status decoder(...);
...
private:
unsigned char       *buffer;
}
```

Marshalling is performed by means of CDR class methods *encoder()* and *decoder()*, based on TypeCode interpretation.

Method *encoder()* is called with a data value, needed to transfer, appropriate TypeCode and pointer to the current position in the octet stream. The algorithm of the method is exactly the same as the interpreter in the introduction of the paper. According to the TypeCode the type of data value is determined. If the data value type is a basic type then appropriate basic primitive put data into the buffer with alignment restrictions. Otherwise, marshalling interpreter examines each of the constituents of complex data values. It does so by making a call to TypeCode::traverse(), and passing itself for future recursive calls. This "visit" function uses the interpreter's knowledge of the environment's size, padding, and alignment rules.

There is a table supporting calculation of size and alignment requirements for any one instance of a given data types. This is indexed via CDR's TCKind values, which are "frozen" as part of the CDR standard. Entries hold either the size and alignment values for that data type, or a pointer to a function that is used to calculate those values. Function pointers are normally needed only for constructed types.

This table has system dependent parts, and so should be examined when porting to new CPU architectures, compilers, and so forth. Issues of concern are primarily that sizes and representations of CORBA primitive data types are correct (key issues are verified when the ORB initializes) and that the alignment rules are recognized.

The method *decoder()* performs the reverse actions and have the same algorithms as *encoder()*, whose differ just by basic primitives 'put' into and 'get' from some buffer. As additional actions decode procedures performs error-checking (e.g. fixed sequence size exceeding) and memory allocation for some variable size data (e.g. *sequence* or *string*).

4. 3. ADVANTAGES AND SHORTCOMINGS OF THE METHOD

The main advantage of the marshalling interpreter is the compact and unified code. As well, it can be said about interpreter technique itself. Interpreter of TypeCode treats data values of any IDL types in the same way. It is important to notice, that interpreter supports marshalling of the types that are not known at compile time. But the TypeCodes of such types can be received from network at run time.

TypeCodes have a number of uses. For example, they are used in the dynamic invocation interface to indicate the types of the actual parameters. So, there was no necessity to think of a new special structure to handle information on types.

Basic primitives are implemented as *inline*-functions to reduce the execution time. However, making marshalling code faster has the drawback of increasing its code size. It can be not desirable for machines with memory size constraints. So, an optimal way of *inline*-substitution usage have to be found [2].

Among the shortcomings of the marshalling interpreter we can point out that it assumes the strict fixed structure of data as it is presented in TypeCode. And if some compiler uses others rules for positioning structure's fields in the memory it could not be treated by interpreter correctly.

The basic shortcoming that is faced by any method based on interpreter technique, is a small execution speed. The loss of productivity is especially noticeable while working with data of deep embedded constructed types due to the mechanism of recursion. Compiled code can help to solve this problem.

5. COMPILED CODE FOR MARSHALLING
5.1. MARSHALLING PROCEDURES

It is obviously, that marshalling interpreter for data of a particular constructed type always performs one and the same algorithm depending on this type. For example, marshalling of any *struct* type consists of consecutive marshalling of all it's fields in the order, described in the IDL specification.

So, we can use compiled marshalling procedures to marshall data of constructed user defined types. Moreover these marshalling procedures can be generated by an automatic tool from a formal IDL specification.

As it has been said above, IDL specification is mapped into particular programming language (C++ in our case). For this purpose IDL compiler is used. The compiler is aimed at the IDL input parsing and translating it into target language, if no errors occur. The translation includes the mapping of IDL types, constants and interfaces onto appropriate target language structures. Also, for each type defined in IDL specification, compiler produces a pair of procedures for marshalling and unmarshalling as well as a set of some other associated procedures.

Marshalling procedures for basic types, containing information on size and alignment requirements for any one instance of a given data types, coincide with correspondent CDR methods that implement marshalling primitive.

For constructed types marshalling procedures are based on the procedures for basic types and have embedded structure. In this way, invocation of needed marshalling procedure performed by means of ordinary procedure call is possible as well as it's *inline*-substitution (not only for basic types as in marshalling interpreter implementation). In our case, *inline*-substitutions are used for compact marshalling methods (few C++ operators) only.

For example, marshalling procedures for type *struct Foo {long l; string s;}* are the following:

```
// encoder procedure
inline CORBA2::Boolean operator <<= (CDR &trg, const FOO &src) {
    return (trg <<= src.l) && (trg <<= src.s); }

// decoder procedure
inline CORBA2::Boolean operator >>= (const CDR &src, Foo &trg) {
    return (src >>= trg.l) && (src >>= trg.s); }
```

Operators <<= and >>= are overridden for each IDL types (basic and user-defined). In the example, at first <<= operator of type *long* and then <<= operator of type *string* are used in *Foo* type <<= operator implementation.

It is interesting to notice, that marshalling and unmarshalling procedures have symmetrical algorithms structures as well as *encoder()* and *decoder()* methods of marshalling interpreter. Right down to some actions special for unmarshalling (memory allocation, error checking).

5. 2. ADVANTAGES AND SHORTCOMINGS OF THE METHOD

Section 6 presents experiment numbers that shows that the speed difference between interpreted and compiled marshalling code is significant. Right now, we just notice that compiled code is consistently faster than interpreted code by a factor of 2 to 5.

As we can expect, we have to pay for making marshalling code faster by increasing its code size. IDL Compiler generates marshalling procedures for each type from IDL specification and those procedures have to be compiled together with others needed procedures and libraries containing ORB functions.

Inline-technique is used to reduce the execution time. As rule, marshalling procedures (for user-defined types too) are exactly *inline*-procedures. It is obviously, that code size directly depends on number of methods and their parameters in IDL specification.

Marshalling interpreter treats “non-typed” data, which structure is described by means of appropriate TypeCode. The problems with fixed internal structure description are faced if a compiler uses another rules for structure’s fields presentation in memory. Type-safe marshalling procedures avoid these disadvantages of marshalling interpreter.

The technique of compiled code for marshalling does not support data which type is not known at compile time. It seams to be the most significant shortcoming of this technique.

6. EXPERIMENT NUMBERS AND THEIR ANALISYS

In order to determine the practical impact of different techniques for marshalling code the following IDL specification is considered:

```
struct S {
    long l;
    string s;
};

typedef sequence <S> Seq;
```

```
struct SC {
    S s[10];
    any a;
    Seq seq;
    string str;
};

typedef SC Arr[10][10];

interface Testing {
    void test_long (in long par);
    void test_struct (in S par);
    void test_seq (in Seq par);
    void test_arr (in Arr par);
};
```

The test consists of consequential invocations of methods declared by i*nterface Testing*. As the types of *in* parameters different IDL types are used.

TABLE 1.

	long	**struct S { long l; string s; }**	**typedef sequence <S> Seq**	**typedef struct SS { S s[10]; any a; Seq seq; string str; } Arr [10] [10]**
compiled code	0.250 ms	0.303 ms	6.011 ms	46.550 ms
interpreted code	0.275 ms	0.470 ms	15.500 ms	199.970 ms
compiled code (C++ optimisation option)	0.155 ms	0.192 ms	2.344 ms	15.680 ms
interpreted code (C++ optimisation option)	0.157 ms	0.255 ms	6.921 ms	92.170 ms

Table 1 gives the absolute throughputs of interpreted and compiled code needed for marshalling of parameters on client and server-side. All absolute numbers were measured on a Sun SPARCstation 4 / microSPARC 100 Mhz, using gcc version 2.7.0.

From these numbers we can see that execution speed of marshalling for data of basic types practically does not depend on code generating technique. In this case, marshalling interpreter and marshalling procedures of basic types perform the same basic primitives without additional actions, specific for each technique.

Compiled marshalling procedures gives great advantage of productivity treating data with deep embedded structure. In this case, interpreter becomes long sequence of recursive calls. But compiled procedures with *inline*-substitution allow to reduce “overhead expenses” for additional calls. For example, the compiled code for marshalling data of *array Arr* type (see Table 1) is faster than interpreted code by a factor of 6, using optimizing option of C++ compiler, and by a factor of 4,5 otherwise.

It is interesting to notice, that compiled marshalling code has better execution speed than usage of a standard optimizing option of C++ compiler for some types (e.g. *sequence Seq* and *array Arr* from the Table 1).

TABLE 2.

	interpreted code	**compiled code**
code size	38408 B	60232 B

However, the code size is increased due to the compiled marshalling procedures, and their *inline*-substitution especially. Table 2 shows the difference in code size between interpreted and compiled marshalling code for some application working with implementation of Naming Service, standard CORBA service. But the goal of generating execution-time optimal code under a size constraint can be met by generating a hybrid between the different implementation technique.

7. CONCLUSION

Marshalling (presentation conversion) is necessary in distributed systems, since modules of those systems use different representation for the same data. The article presents two alternative implementation technique for marshalling in the context of Object Request Broker design.

An experimental evaluation of interpreted and compiled code for marshalling confirms the well-known size-speed trade-off.

For ORB design the execution speed is most important character, so we show a preference for compiled code technique. But sometimes (on a machine with memory size constraints) a more flexible code generation strategy can be required. Instead of using the same strategy for all types, different code generation strategies (interpreted code, compiled code and inline code) should be used for different types. For example, for a type definition node in syntax tree, the compiler implemented such “hybrid” strategy must decide whether interpreted or compiled code should be generated. For a field node, the compiler must decide whether the marshalling routine for the field should be written inline.

The selection of a code generation strategy for a particular type depends on the following factors. The frequency with which the type occurs at run-time. The execution times of the marshalling routines. The code size increase incurred by each of the alternatives.

The principle of locality (Knuth, 1971) says that programs spend a large part of their execution time in a small part of the program code. Analogues researches [2] have shown that this principle also holds for communication code. So, large performance improvement of marshalling code can be achieved by optimizing only a small part of the implementation (e.g. only marshalling procedures of the types that occur frequently).

It is an IDL specification, that contains hints which part of it will be executed frequently. Intuitively, it seems clear that types that occur as field of an array will occur frequently, that types which are referenced frequently by other types will also occur frequently. Such static approach can be successfully used for finding IDL specification parts that can benefit from code optimisation [2]. This approach does not depend on particular application working with an IDL specification. So it can be realized by IDL compiler itself.

On the other hand, for particular application, frequency of type usage can be achieved more precisely by means of program profilers. Program profile taking into account run-time peculiarity, such as loops and branches, shows variables of which types take part in communication more frequently. An optimizing IDL compiler uses this statistical information to select one of several code generating alternatives for each node of syntax tree. Also, IDL compiler can make more informed optimisations based on domain-specific heuristics.

So, to achieve size-speed balance we can optimize marhalling implementation only for a part of IDL specification. There are several interesting areas for future research: algorithm of finding IDL specification parts that can benefit from code optimisation, the development of appropriate IDL compiler.

REFERENCES

1. The Common Object Request Broker: Architecture and Specification. Revision 2.0. July 1995. Object Management Group.

2. Philipp Hoschka. Automating Performance Optimisation by Heuristic Analysis of A Formal Specification. IFIP TC 6 / 6.1 International Conference on Formal Description Techniques IX (Theory, application and tool), 8 - 11 October 1996, pp. 77 - 92.

3. Hiroki Horiuchi, Tetsuya Kuroki, Sadao Obana, Kenji Suzuki. EPER: Efficient Packed Encoding Rules for ASN.1. IFIP TC 6 / 6.1 International Conference on Formal Description Techniques IX (Theory, application and tool), 8 - 11 October 1996, pp. 179 - 194.