Marshaling/Demarshaling as a Compilation/Interpretation Process

Christian Queinnec
Université Paris 6 — Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex — France
Christian.Queinnec@lip6.fr

Abstract

Marshaling is the process through which structured values are serialized into a stream of bytes; demarshaling converts this stream of bytes back to structured values. Most often, for a given class of data, the marshaler and the demarshaler are tightly related pieces of code that are synthesized conjunctly. This paper proposes a new point of view: the demarshaler is considered as a byte-code interpreter *evaluating a stream of bytes that is itself considered as a* program *i.e., as a sequence of commands interspersed with quoted raw data. These programs are expressions of the* marshaling language. *From that point of view, the marshaler logically appears as a* compiler *translating structured values into expressions of the marshaling language.*

The demarshaler depends on the sole marshaling language. If this language is powerful enough to deal with any kind of data then the demarshaler can be kept constant while many marshalers may coexist. This asymmetry and programmatic view has far-reaching consequences: (i) *it is simple to accommodate new dynamically created classes of data,* (ii) *it is possible to have simultaneously various marshalers that offer different, even evolving, strategies in order to cope with different situations such as network congestion or processor memory exhaustion.*

1. The Context

According to the taxonomy of Distributed Shared Memory [5], DMEROON is a library of functions allowing objects to be shared over the Internet in a coherent way among multiple readers and, at most, one writer. Management is distributed and coherency is causal. Coherency is out of the scope of this paper [6] and so are many details of DMEROON [7]. The DMEROON project started as the memory layer supporting a distributed language [9] but it soon tried to be a portable common layer for different languages (currently C and Scheme) to exchange, copy or share typed structured values.

Sites are autonomous address spaces connected by messages. An *object* is a contiguous chunk of bytes belonging to a single site. An object is made of *fields*. A *regular field* holds a single value of any legal C type. An *indexed field* holds a contiguous sequence of homogeneous values of the same C type (size and alignment constraints are those of C) prefixed by a number recording the length of the sequence (all unsigned C types are possible for these lengths). Lengths of indexed fields are determined at allocation time. Indexed fields permit representation of strings (indexed field of characters), (Java) vectors (indexed field of pointers) and even activation frames (Corba's NVList for instance) as regular, mundane, plain objects. For any object, it is possible to obtain the values that are held in its fields and to mutate these values provided the field was declared *mutable*. Moreover, for any object, it is possible to access *structural (reflective) information* such as the length of its indexed fields or, its *class* from which may be retrieved the layout of its fields.

Semantically, pointers raise questions about the identity of objects, whether they are shared or copied between sites, what kind of coherence is achieved if they contain mutable fields, etc. Independently of these questions the DMEROON memory model simply supports, at the most fundamental level, the concept of *remote pointer* allowing an object from one site to refer to another object of another site.

An object is *owned* by the site where it was created reciprocally, the object is said to be *local* to that site. A remote object (i.e., an object that is not local) may be locally cached with a *replica*. The remote object is then said to be *present* since its replica stands for it. A remote pointer to a present object may be swizzled [4] into a local pointer to the replica. A local object is of course a present object. Objects may migrate i.e., may change their owner and still keep their identity. A companion report [8] presents the details of the remote pointer machinery.

Albeit apparently simple, handling remote pointers is rather subtle. One reason is the different sets of invariants the implementation has to maintain with respect *(i)* to the user or, *(ii)* to itself or, *(iii)* to the other sites. Another difficulty is the number of simultaneous goals this machinery, and particularly the DMEROON machinery, also wants to achieve: fast dispatch for object-oriented programming, support for garbage collection, reflection (access to the structural information), coherence (between an original object and its replicas), dynamicity (creation of new classes), mobility (of objects), security or fault-tolerance.

Sites are connected through an *object pipe*. When an object is written in such a one-way pipe, another structurally "equivalent" object is created at the other end. It is up to the higher layers of the library to ensure coherency between equivalent objects: for instance, such objects have to be told that they are replicas of remote objects in order to obey some coherency protocol. The object pipe is an interesting layer that hides the message machinery. Requests from one site to another are, for example, represented by objects. The answer to a request just refers to the original request with a (remote) pointer. Forwarding a request leaves a trail of pointers that may be followed back by the answer.

C types other than pointers do not pose much problem if sites agree on a common representation as did XDR [11], CDR [10], or the Java serialization protocol. Remote references are encoded with an unique non ambiguous key (as well as some other information, an IOR in Corba's parlance) allowing sites to retrieve them. But to make the object pipe useful and efficient: *(i)* the marshaler has to reduce the number of messages i.e., objects should be sent (pushed) in advance but not too much (the receiving site must not be overwhelmed by too many objects), *(ii)* the compiler must not loop while marshaling cyclic data, *(iii)* resources allocated to marshaling should be under control, *(iv)* some network invariants should be enforced.

The point of this paper is to present a programmatic view of marshaled objects. The demarshaler is a byte-code interpreter that reads bytes from the object pipe and executes them to build equivalent objects. The marshaler just appears as a compiler transforming partial graphs of objects into a stream of bytes, a sequence of raw XDR bytes structured by demarshaling commands. Once the marshaling language is set, the demarshaler derives simply from it. However to have only one marshaler does not preclude a wide variety of marshalers to be imagined, written, tested. With this linguistic view it is possible as well as easy to change marshalers at run-time to react to new network conditions. New classes of objects may bring their own marshaling techniques. Marshalers may also be disseminated via active network à la [2].

This paper only focuses on the sole marshaling/demarshaling process. The marshaling language appears in Section 2; a naive marshaler is commented in Section 3 and extended in Section 4. Related work concludes the paper.

2. The Marshaling Language

Sites are connected via an object pipe that conveys marshaled objects i.e., programs, conforming to the marshaling language. This Section presents the basic six primitives of this very specific language that deals with distributed objects in order to allocate them, manage their replicas, update them etc. The marshaling language is expression-oriented. Its most primitive commands appear in Table 1.

The `allocate` primitive expects at least one argument: a class. Since a class contains all the structural information that characterizes its instances and since this class is present (this is why it appears underlined in Table 1), it is simple to determine the number of its indexed fields and to expect as many sizes, of the proper unsigned C type, as additional arguments of the `allocate` invocation. The value of this primitive is the allocated object with complete structural information: this object is local to the site where the `allocate` primitive is run.

The `fill` primitive expects an object; this object must be present. The class of this object is used to consume the subsequent bytes until the object is entirely filled. Non-pointer fields are encoded similarly to XDR. Pointers are encoded as objects i.e., as expressions of the marshaling language. The filled object is returned as the value of the `fill` expression.

The two previous primitives relies upon the following *network invariant*:

No object can be demarshaled if its class is not present.

There is a simple way to respect the previous network invariant (a different solution will be presented in Section 4 with the `try` marshaling command): a site never marshals an object whose class may be unknown by the receiving site, instead the receiving site will receive a remote pointer from which it will have to pull the class before pulling the object (of course, the user is not aware of that, the user simply asked for the object at the end of a pointer).

A site is cited via the `site` primitive. This primitive expects host and port numbers. A site is represented by an object that concentrates the information that is common to all objects of that site. No communication is required with the mentioned site.

Remote references are created with the `remote` primitive. This primitive expects a key, a site and a class. The site must be present (so enough information is present to create a communication channel if needed) but the class may be remote (enough information is present to fetch it if needed). If the remote reference already exists on the receiving site it is returned (and shared) otherwise it is built afresh. Observe that the class of the remote object is available, this eases to respect the afore-mentioned network invariant.

is called as	does	returns
`allocate` *<u>class</u> sizes...*	Allocate an object	the allocated object
`fill` *<u>object</u> content...*	Fill an object with some content	the filled object
`site` *IP port*	Refer to a site	the site
`remote` *key <u>site</u> class*	Refer to a remote object	the remote object or its replica if present
`bind` *<u>object</u> remote-object*	Associate a replica to a remote object	the replica of the remote object
`predefined` *index*	Refer to an ubiquitous object	the index'th ubiquitous object

Table 1. Basic primitives of the marshaling language (underlined arguments require the corresponding objects to be present on the demarshaling site.)

The `bind` primitive normally expects a local object and a remote object; it makes the local object become the replica of the remote object. Of course, the remote object should not already have a local replica.

Finally, there may exist some ubiquitous (often immutable) objects (booleans, predefined classes, predefined sites, etc.) that exist on every site. They are specially encoded for compactness reasons.

Most of these primitives are unsafe or dangerous. They all err when an argument that should be present is not. The `fill` primitive allows objects to be overwritten, the `remote` primitive may confer an inappropriate class to a remote object, the `bind` primitive may associate unrelated objects. It is up to the marshaler and the upper layers (see Table 6) to ensure safety with respect to representational invariants (see [8] for details).

3. A Simplistic Marshaler

The marshaler translates objects into expressions of the marshaling language. However, the compilation is not so obvious since: *(i)* to reduce the number of messages, objects should be pre-sent (i.e., pushed) but not too much: the compilation ought to stop. *(ii)* the compiler must not loop while marshaling cyclic data, *(iii)* last, the network and representational invariants must be enforced.

A simple compiler may be explained as the conjunction of some smaller naive compilers. The $\mathcal{C}_{\text{predef}}$ compiler just takes care of ubiquitous objects and emits accordingly `predefined` commands. The $\mathcal{C}_{\text{share}}$ compiler never sends objects but sends instead remote references onto them (with `remote` commands), this compiler always try to share objects hence its name. At the opposite is the $\mathcal{C}_{\text{copy}}$ compiler that always send objects (an `allocate` embedded in a `fill` command). Between these last two compilers, the $\mathcal{C}_{\text{presend}}$ compiler sends objects if predefined otherwise it shares them but try to pre-send their content if their classes are predefined.

Compilers are ordered by the level of sharing they provide, they are ranked as follows: $\mathcal{C}_{\text{copy}}$, $\mathcal{C}_{\text{presend}}$, $\mathcal{C}_{\text{share}}$, $\mathcal{C}_{\text{predef}}$. Whenever an object is marshaled with a compiler, the objects it refers to are marshaled with a more sharing compiler. This order ensures the termination of the compilation but other methods are obviously possible.

The resulting compiler is simplistic but satisfies the network invariant.

4. Enriching the Marshaling Language

The marshaling language is very raw for the moment and restricted to a few primitives. But to view it as a language helps to make it better to *(i)* obtain more compact marshaling programs, *(ii)* diminish the number of messages, *(iii)* keep control of resources (stack, cache) devoted to marshaling.

The marshaling language is expression-oriented, it is therefore straightforward to extend it with a stack and operations on that stack. We therefore add the primitives of Table 2. This Table may be completed with other Forth-like operations such as `swap`, `roll`, etc. Observe that a stack is specific to a one-way object pipe. It exists on the emitting site; the receiving site maintains a copy of that stack which is similar to the one of the emitting site up to the commands that are not yet demarshaled.

is called as	does	returns
`push` *object*	push an object	the pushed object
`top`		the top of the stack
`pop`	pop the stack	the popped object

Table 2. Stack marshaling commands

These operations are useful in at least two situations: *(i)* when marshaling many times a same object (for instance, an object used to initialize the cells of a vector) *(ii)* when marshaling short cycles forming a mutually recursive data (this is simple to handle since allocation and filling are two distinct operations).

A small example may be in order. We can marshal the vector and the integer of Figure 1 into the following expression (where parentheses were added for clarity):

```
(fill (allocate Vector 2)
```

```
(push (fill (allocate Integer)
            42 ))
(pop) )
```

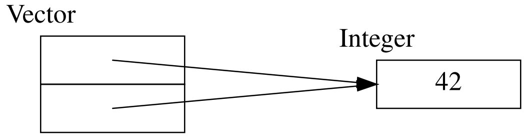

Figure 1. A vector initialized with a shared value.

A stack helps for a single message, a cache may help for a series of messages. The sent objects may be inserted in a cache and later referred to with only a few bytes. Instead of providing a static caching strategy, we prefer to enrich the marshaling language with a new set of commands to manage this cache, see Table 3. Observe that a cache is specific to a one-way object pipe. It has one occurrence on the emitting site, the receiving site maintains a copy of that cache which is similar to the one of the emitting site up to the commands that are not yet demarshaled.

A good caching policy probably depends heavily on the user's applications requirements. The availability of the caching commands allow to design new appropriate marshalers with innovative, adaptive caching policies.

In order to reduce the size of messages, DMEROON also uses the following commands, see Table 4. They don't introduce new concepts, they are pure but very common abbreviations. They allow to refer compactly to objects owned by the sites at both ends of an object pipe. These abbreviations drastically reduce the size of messages since most of the objects that are exchanged between two sites are owned by either one of these two sites. However they have an impact since their use makes messages non portable (messages must be demarshaled then remarshaled in order to be routed towards another site) since the result of the marshaler depends on the sites that are at the ends of a communication channel.

Besides the previous commands, DMEROON adds three more technical commands, see Table 5. The two first commands, `prog1` and `prog2`, allow expressions to be gathered for their side-effects (stack or cache commands are clearly candidates).

is called as	**returns**
`prog1` *`object object`*	the first object
`prog2` *`object object`*	the second object
`try` *`size object object`*	the first or second object

Table 5. Ancillary marshaling commands

With these additional primitives, a possible marshaling of the vectors of Figure 2 is:

```
(prog1 (record 47 (allocate Vector 2))
(prog1 (record 48 (allocate Vector 1))
(prog1 (fill (refer 48) (refer 47))
       (fill (refer 47) (refer 48)
                        (refer 48)))))
```

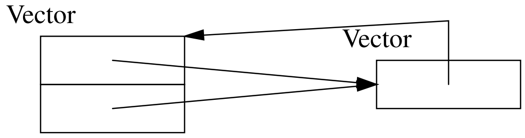

Figure 2. A small cycle

The most interesting command is the third of Table 5. The `try` command tries to demarshal its first object (appearing as second argument). If demarshaling this first object is free of errors then `try` acts similarly to `prog1` and returns this first object. If an error occurs while demarshaling the first object, then `try` skips it and behaves similarly to `prog2` that is, it returns the second object (appearing as its third argument). Wherever is the demarshaling error, it is always possible to skip the first object since the length of its encoding is available in the first argument of `try`.

The `try` command is powerful since it confines demarshaling anomalies, it also allows new strategies that still respect the network invariant. One may try to send an object assuming that its class is known from the receiving site but, to play it safe, the remote pointer is also sent to allow the receiving site to fix the problem of the missing class. In other words, one may send an object o with the $\mathcal{C}_{\text{try}}$ compiler defined as:

$$\mathcal{C}_{\text{try}}(o) \mapsto \texttt{try} \quad n \quad \underbrace{\mathcal{C}_{\text{copy}}(o)}_{n\ bytes} \quad \mathcal{C}_{\text{share}}(o)$$

If the content of the object o cannot be demarshaled then this expression simply returns a remote pointer onto o.

The `try` command is obviously reminiscent of the `try` or `unwind-protect` keywords of well-known programming languages and actually comes from the point of view we adopted for the marshaling language.

The semantics of the marshaling language could have been presented to describe more precisely the formal meaning of these commands and their interaction. For instance, a `try` command resets the stack at the height it had when `try` started but the cache is left unchanged. With this semantics, one may prove whether a marshaler respects the representational invariants.

is called as	does	returns
`record` *index object*	record an object with a given index	the object
`refer` *index*		the index'th object of the cache
`double`	double the size of the cache	nothing
`reset`	empty the cache	nothing

Table 3. Cache marshaling commands

is called as	returns
`receiving-site`	the receiving site
`emitting-site`	the emitting site
`emitter-reference` *key class*	≡ `remote` *key* `emitting-site` *class*
`receiver-reference` *key*	≡ `remote` *key* `receiving-site` <*the appropriate class*>

Table 4. Abbreviation marshaling commands

5. Related Work

The XDR (for eXternal Data Representation [11]) library, introduced by the NFS system, allows to marshal structured values. This is a lower level library since it does not deal with (remote) references nor it does introduce the illusion of a distributed memory model (no object identity). The marshaler and the demarshaler form a single piece of code whose behavior is specified at invocation time. This code is generated from the description of the data structure, typically a `.h`-like file. Messages are shaped after the control structure of the generated de/marshaler.

Corba introduced CDR (for Common Data Representation [10]) for marshaling. Since Corba brings the notion of object identity, references to remote objects are easily marshaled. However these libraries are opaque, cannot be tailored, and, as for XDR, statically generated from the description of the exchanged data structure expressed in IDL (for Interface Definition Language). However Corba also offers a Dynamic Invocation Interface (DII) to cope with non static situations.

Static generation of marshalers/demarshalers produces a code whose size may be extremely large since it depends on the number of classes. Some alternate solutions were explored. It was proposed some years ago to interpret type descriptors and shown that the speed is not too much deteriorated since the demarshaler is very compact and fits well in processor cache. [1] also proposed a kind of interpreter that he called *the marshaling engine*. Finally, [3] lessens the need for space with just-in-time stub generation.

Within the Java realm, RMI (for Remote Method Invocation) introduces a serialization/deserialization interface. This interface takes care of all sorts of objects (provided they are `serializable`) and may be customized or extended by the user. The caching policy cannot be parameterized: by default, all sent objects are memorized.

Compared to these proposals, ours is clearly more compact: On a PC box with Linux, the DMEROON demarshaler weighs 18 Kbytes. The marshaler, which is slightly better than the naive one above, adds 6 Kbytes. These sizes are independent of the number of classes although the 103 predefined DMEROON classes add some 17 Kbytes. Were we to use `rpcgen`, these classes will generate a static marshaler/demarshaler of 23 Kbytes to which we must add the necessary XDR library making up to a total of 79 Kbytes (not taking into account DMEROON indexed fields which are not naturally accommodated by `rpcgen`).

With respect to speed, our solution is clearly slower than XDR-style compiled code although the whole process is clearly dominated by network latency. Every demarshaler has to decipher (interpret) an incoming stream of bytes. Conversely to XDR where description of types is compiled, our solution interprets type descriptors. The flexibility of this compilation/interpretation process allows the marshaler to be enriched at run-time to incorporate new dynamically created classes or user's dynamically specified customization. The marshaler may also react to overall changes such as network bandwidth, memory exhaustion etc. Marshalers can themselves be marshaled and disseminated over the network. Our solution is portable, does not depend on the operating system and tolerates a Garbage Collector recycling unused classes for instance.

Compared to dynamic situations (Corba's DII for instance), our solution offers some advantages: it may use just-in-time technology and be compiled (or partially evaluated) to gain speed. Since we extend XDR, part of the XDR library may be woven into our marshalers, this will mainly benefit to flat numeric matrices whereas our technique is better suited to marshal parts of graphs of linked objects. Moreover our solution may also be used when marshaling is done via XML [12] i.e., structured plain text.

6. Conclusions and Future Work

The main point of this paper is to consider the stream of bytes carrying marshaled objects as a sequence of expressions of some marshaling language. This programmatic view adds more meaning to the exchange protocol, stresses the asymmetry between the marshaler (a compiler) and the demarshaler (an interpreter) and, brings all useful language technologies to the level of marshaling. The rest of the paper exposes some corollaries. The marshaling language is inspired by programming languages and allows for versatility both at compile-time and run-time.

The paper is supported by the layered architecture of Table 6.

Application Programming Interface	user
Request/Answer protocol	protocol between sites
Marshaling language	commands
Representational properties	invariants
Memory model	objects, classes, replicas

Table 6. Layers

We think that the few allusions at formalism here and there in this paper may be a hint for some proof systems, at the interface between layers:

- to prove the semantics of the marshaling language not to violate representational invariants,
- to prove a marshaler with respect to *(i)* the fundamental network invariant or *(ii)* other user-oriented invariants,
- to prove a protocol over the marshaler to respect some network invariants.

We plan to develop these points as well as to experiment with the reification of marshalers in order to let class conceptors express their marshaling needs.

These ideas have been implemented in the DMEROON distributed shared memory since 1996. Additional details may be found in the DMEROON documentation available from:

`http://www-spi.lip6.fr/~queinnec/WWW/DMeroon.html`

References

[1] A. Bartoli. A novel approach to marshalling. *Software Practice and Experience*, 27(1):63–86, Jan. 1997.

[2] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A packet language for active networks. In *ICFP '98 – International Conference on Functional Programming*, pages 86–93, 1998.

[3] M. Hof. Just-in-time stub generation. In *JMLC'97 — Joint Modular Languages Conference*, pages 197–206, Linz (Austria), Mar. 1997.

[4] J. E. B. Moss. Working with objects: To swizzle or not to swizzle? Technical Report 90–38, University of Massachusetts, Amherst, Massachusetts, May 1990.

[5] J. Protić, M. Tomašević, and V. Milutinović. A survey of distributed shared memory systems. In *Proc. 28th annual Hawaii International Conference on System Sciences*, volume I (architecture), pages 74–84, 1995.

[6] C. Queinnec. DMEROON: Overview of a distributed class-based causally-coherent data model. In T. Ito, R. H. Halstead, Jr, and C. Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, Lecture Notes in Computer Science 1068, pages 297–309, Beaune (France), Oct. 1995.

[7] C. Queinnec. DMEROON *A Distributed Class-based Causally-Coherent Data Model – General documentation*. LIP6, 1998. Rapport LIP6 1998/039 <http://www.lip6.fr/reports/lip6.1998.039.html>.

[8] C. Queinnec. Marshaling/unmarshaling as a compilation/interpretation process. Research Report LIP6/1998/049, LIP6, Dec. 1998.

[9] C. Queinnec and D. De Roure. Design of a concurrent and distributed language. In R. H. Halstead Jr and T. Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachussetts USA), Oct. 1993.

[10] J. Siegel. *Corba, Fundamentals and Programming*. John Wiley and Sons, 1995.

[11] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, Mar. 1989.

[12] World Wide Web Consortium. Xml. http://www.w3c.org/.