Marshaling/Unmarshaling as a Compilation/Interpretation Process

Christian Queinnec

Marshaling/Unmarshaling as a Compilation/Interpretation Process

Christian Queinnec*LIP6 & INRIA-Rocquencourt

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

The unmarshaler depends on the sole marshaling language. If this language is powerful enough to deal with any kind of data structures then the unmarshaler can be kept constant. This has far-reaching consequences: *(i)* it is possible to manage new dynamically created data structures, *(ii)* it is possible to have various marshalers compiling values along different, even evolving, strategies in order to cope with different situations such as network congestion or processor memory exhaustion.

These ideas have been satisfactorily embodied in the DMeroon distributed shared memory since 1996, [Que95, Que98]. DMeroon supports meta-classes, reflective description, dynamic creation of classes, lazy propagation of classes, among other properties. The present paper extends [Que97] (in French) in several ways: it is less allusive, more formal and abstract, and its results are more amenable to be re-used in comparable systems.

The paper is in two layers. Section 1 is the first layer and presents the details of the DMeroon memory model, how remote references and object identities are managed as well as the representational invariants that are maintained at various levels (user-, protocol-, implementation-). Although supported by known techniques, this Section is an accurate account of a quite subtle and rarely described machinery. This memory model fits the goals of distribution but may as well be used to ensure file-based persistency.

The second layer presents the marshaling technique and how it fits with the supporting machinery. More precisely, the marshaling language appears in Section 2; a naive marshaler is commented in Section 3; extensions are detailed in Section 4 while variant marshaling strategies are described in Section 5. Related work conclude the paper.

1 The Memory Model

According to the taxonomy of Distributed Shared Memory, see[PTM95], DMeroon is a library of functions allowing to share objects over Internet in a coherent way among multiple readers and, at most, one writer. Management is distributed and coherency is causal. This Section will focus on the implementation of objects and remote pointers. Coherency is out of the scope of this paper [Que95, Que94] and so are the details of DMeroon [Que98]. The DMeroon project started as the memory layer for a distributed language [QD93] but soon tried to multi-lingual i.e., be a common layer for different languages (currently C and Scheme) to exchange, copy or share typed structured values.

1.1 Objects

Sites are autonomous address spaces connected by messages transmitted through *communication channels*. An *object* is a contiguous sequence of bytes belonging to a single site. An object is made of *fields*. A *regular*

*Université Paris 6 — Pierre et Marie Curie, LIP6, 4 place Jussieu, 75252 Paris Cedex, France – Email: Christian.Queinnec@lip6.fr This work has been partially funded by GDR-PRC de Programmation du CNRS.

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 allow to represent mundane objects such as strings (indexed field of characters) or (Java) vectors for instance. 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 information* such as the length of its indexed fields or, its *class* i.e., its type descriptor from which may be retrieved its field descriptors.

C types other than pointers do not pose much problem if sites agree on a common representation as did XDR [Sun89], CDR [Sie95], or the Java serialization protocol. 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 are mutable, etc. Independently of these questions which depend on the semantics of the language operating DMeroon, the memory model we propose simply supports, at the most fundamental level, the concept of *remote pointer* allowing an object from a site to refer to another object of another site. A remote pointer may be swizzled into a local pointer if the target of the pointer is locally present.

Albeit apparently simple, the machinery of 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 also wants to achieve: fast dispatch for object-oriented programming, support for garbage collection, reflection, coherence, security or fault-tolerance. This paper mainly focuses on the handling of remote pointers.

1.2 Classes

Table 1 summarizes the fundamental classes we will rely upon. This Section comments them and sets up some terminology.

Site	**Class**	**Field**	**Entry**	**Exit**
IP, port	name	name	key	siteof
exported	supers	nature	object	key
exits	fields	C type		replica
sites	properties	properties introducing		classof

Table 1: Fundamental classes

Site holds the information that characterizes an address space: **Site** reifies sites. Apart the IP host and port numbers, there are the *exported* and *exits*[1] tables as well as the table of known *sites*. This latest table gathers the replicas of sites that own the remote objects a given site is aware of. These tables are described below.

Class is the class of descriptors of data structures. A class has a name[2], some super-classes and refers to the fields that its instances contain. **Field** describes a field with a name, a nature (regular or indexed), a C type and some properties such as mutable or immutable. A field also refers to the class that introduces it. As in ObjVlisp [BC87], classes are instances of **Class** and our model supports to subclass **Class** or **Field**. The predefined classes of Table 1 are immutable and *ubiquitous*: they are present everywhere.

Classes, fields and sites are objects the user may freely obtain and use. On the other hand, tables are internal values that may solely be used by the implementation and so are *proxies* (i.e., **Entry** or **Exit** items according to the terminology of [LQP92]).

After creating an object within a site, an user may access its fields or get its class with the `.class` operation. The current site is said to *own* the freshly created objects (migration of objects i.e., change in the owning site is beyond the scope of this paper). Reciprocally the object is said to be *local* to its owning

[1] This asymmetry of names is explained later.

[2] This name is used for human readability. There is no primitive way to retrieve a class from its name.

site. The owning site may be obtained with the `.site` operation. The `.class` and `.site` operations are mere notations, they do not require objects to implement such fields.

Besides a class and a site, an object is possibly associated to a proxy[3]. A proxy is accessed with the `.proxy` operation. Proxies hold the information required to *(i)* let the object be known from other sites or, *(ii)* access its remote content. No proxy means that the object is not known from other sites. An ubiquitous object is local to every site.

1.3 Actions

As far as distribution is concerned, the user may perform three actions only. Given an object o owned by a site s, the user may:

1. create, on a site s', a remote reference towards o. This occurs when an user "sends" an object towards a remote site i.e., lets this object be known from another site.

2. create a *replica* of o on site s'. This corresponds to a copy of o on s'. The access to the replica may of course be constrained by some coherency policy if, rather than just copied, the original object must be shared or synchronized.

3. re-export a remote reference onto another site s''. This is a kind of indirection from s'' to s passing through s' as a relay site. Some systems [SDP92, MDF97] shortcut remote references to be always direct; others [Piq91] maintain the diffusion tree of objects. Both behaviors are supported by DMEROON.

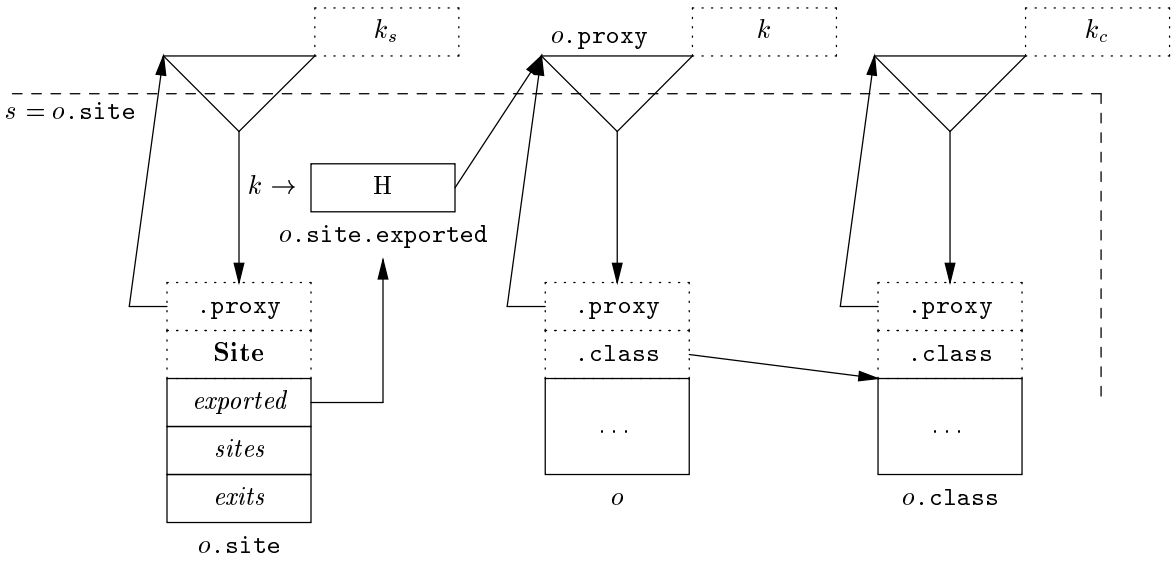

Figure 1: Externalized object on s. The Figure is centered around object o and all details are not shown. The entry items are shown as wedges entering the site bordered with a dashed line. The arrow starting from an entry item designates the object held in the `.object` field of this entry item; the key is shown in the dotted box nearby the entry item. The site s is externalized, its class (**Site**) is ubiquitous. The class of the object o is not ubiquitous, it is externalized (its class may be **Class** or another indirect instance of **Class**).

The user may send to another site a reference to one of its local object. In order to let an object be known from other sites, first, the object is *externalized* (see Figure 1): an associated **Entry** item is created and inserted in the exported table of the current site. This entry item holds a unique key that identifies the

[3]Currently, the DMEROON system provides two hidden pointers per object to hold its class and proxy, the site being left implicit.

object network-wide and a pointer to the object. Reciprocally, the externalized object holds this entry item as proxy so at most one entry item is associated to an object. By construction, the user never sees entry items although the implementation needs to access them via the `.proxy` operation. Observe that the class of the object is also externalized. The instance of **Site** that describes the current site is always externalized as a result of the establishment of communication channels with other sites.

After the object is externalized on site s, a remote reference may be created on a site s' as follows (see Figure 2): an **Exit** item is created and inserted into the exits table of s'. This exit item holds a key (the key of the matching entry item) as well as a pointer to the local instance of **Site** which describes s on s'. This replica of site s is only used for its IP host and port numbers[4]. An object which is not local is said to be *remote*.

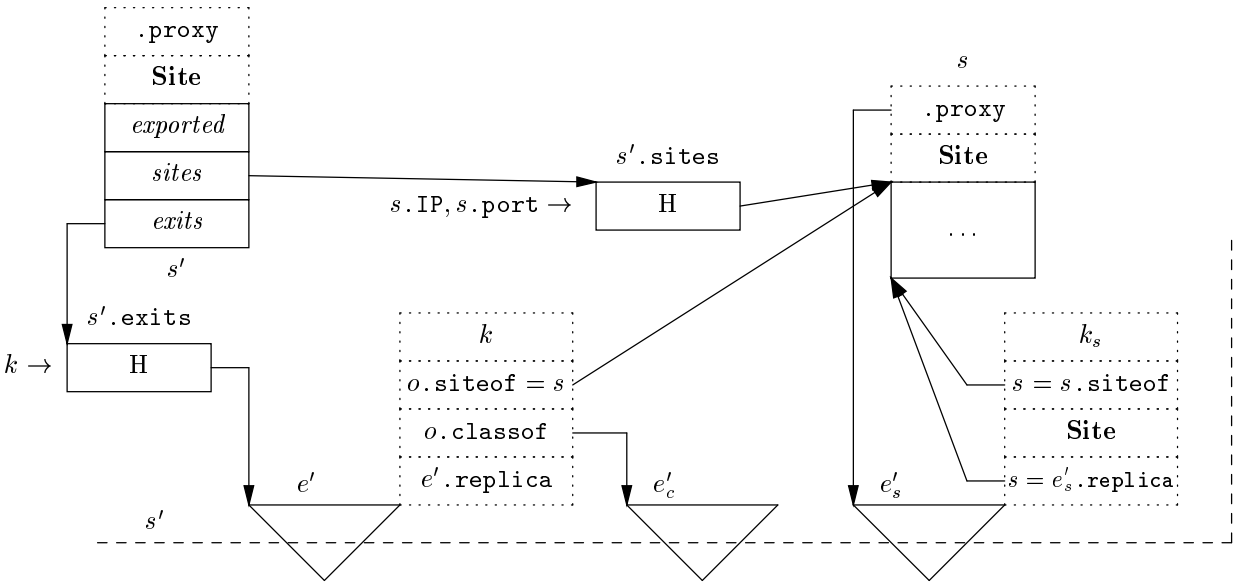

Figure 2: Remote pointer from s'. All details are not shown. Exit items are represented by wedges exiting out of the site bordered with a dashed line. Object o is remote on s' and currently has no replica. Object o is represented by the leftmost exit item e'. Its class is also remote and has no replica either. Site s has a replica; its class, **Site**, is ubiquitous therefore present and local. The site of s is, of course, s itself. Sites are immotile, they cannot be migrated.

An exit item also holds two additional fields that characterize the remote object. The `.classof` field holds a (possibly remote) pointer to the class of the remote object, see Figure 2. The second additional field of an exit item either is `NULL` or holds a *replica* of the remote object. When the user dereferences a pointer to a remote object, a replica is fetched and returned to the user. A replica allows to speed up the accesses to the remote object it stands for, see Figure 3. Of course, if such a replica exists, its structural information is similar to the structural information of the remote object: the replica and the remote object have same owning site, same class and same lengths for indexed fields. However the content of the replica may be whatever the coherence protocol decided it to be. A replica is always associated to an exit item and has it as proxy.

When a replica is present on a site, any pointer to any exit item may be *shortcut* (or swizzled [Mos90]) to point directly to the associate replica if it exists. Conversely, the GC may *longcut* a reference to a replica and redirect it towards its associated exit item so the GC is able to recycle the replica if useless. These two rules may be applied at any time, see left part of Figure 3. These mutations may only be performed on the pointers that occur in the fields of objects. Structural pointers required by the implementation to refer to the class, the site, the proxy or the replica are not submitted to this rule; this is to avoid infinite regression within the implementation.

[4] This is easily achieved in DMeroon since all the other fields of **Site** are declared *local* and *secret* so they are never marshaled out of the owning site nor they may be accessed by the user.

$\forall o \in \textbf{Object}, \forall e' \in \textbf{Exit}, \forall f \in o.\texttt{class.fields}$,
$e'.replica \neq \texttt{NULL} \Rightarrow$

$$o.\texttt{f}{:=}e' \underset{\text{longcut}}{\overset{\text{shortcut}}{\rightleftarrows}} o.\texttt{f}{:=}e'.\texttt{replica}$$

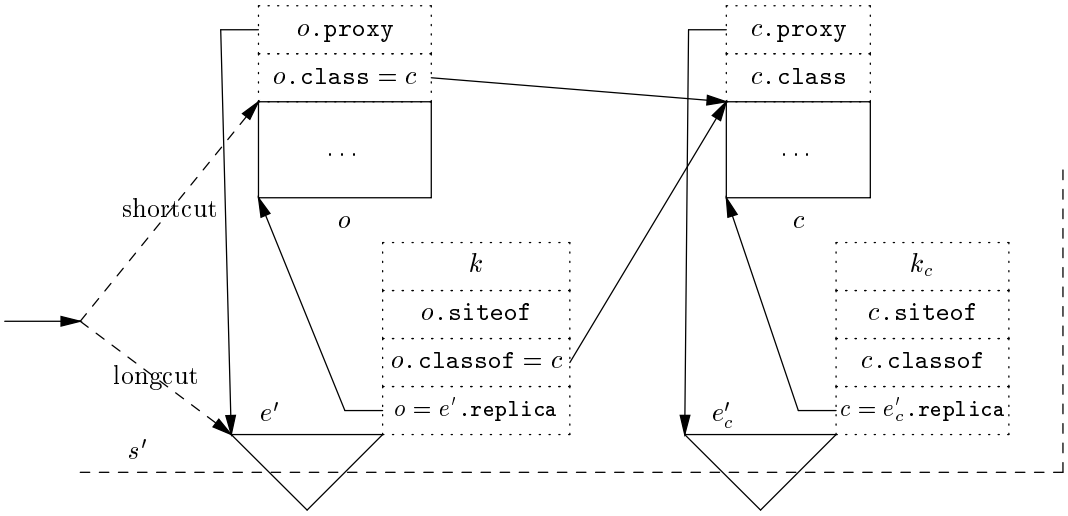

Figure 3: Replicated object. All details are not shown. Object o is remote on s' but has a local replica (therefore its class c is also present i.e., has a local replica). A pointer to o may refer indifferently to the exit item e' or the replica o.

Finally and since it is often useful to re-export a reference to an exit item even if the remote object is not present, we allow to *re-export* an exit item i.e., to insert it in the exported table, see Figure 4. This table is named *exported* rather than, say, *entries* since it may contain entries or exits items. Since an exit item already has a key, it is externalized under that same key.

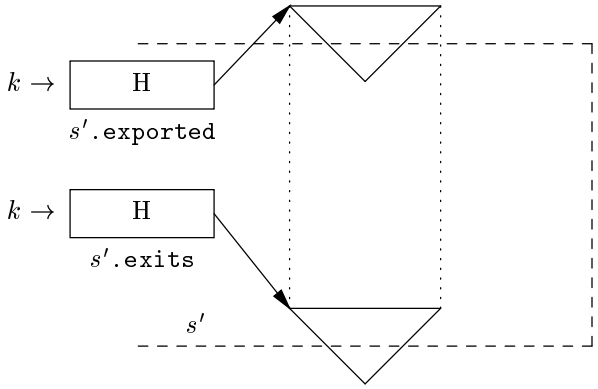

Figure 4: Re-exported remote object. The re-exported exit item is drawn twice (as an in-going and out-going wedge) but forms a single entity. A re-exported exit item needs not be associated to a replica.

At the implementation level, if we consider the `.replica` field of the exit item to be an alternate name for the `.object` field of the entry item, then an exit item may be viewed as a subclass of an entry item in object-oriented parlance.

1.4 Representational Properties

This Section presents a more formal and synthetic view of the representational properties, they are sketched in Table 2 and commented below.

$\forall o \in \mathbf{Object}, \forall s, s' \in \mathbf{Site}$,

$local_s(o) \stackrel{\text{def}}{=} o.\texttt{site} = s \wedge present_s(o.\texttt{class})$

$externalized_s(o) \stackrel{\text{def}}{=} local_s(o) \wedge \exists k, is\text{-}externalized_s(o,k) \wedge exported_s(o.\texttt{class}) \wedge exported_s(o.\texttt{site})$

$is\text{-}externalized_s(o,k) \stackrel{\text{def}}{=} \exists e \in \mathbf{Entry}, o.\texttt{proxy} = e \wedge s.\texttt{exported}(k) = e \wedge e.\texttt{key} = k \wedge e.\texttt{object} = o$

$remote_{s'}(o) \stackrel{\text{def}}{=} \exists k, is\text{-}remote_{s'}(o,k)$

$$is\text{-}remote_{s'}(o,k) \stackrel{\text{def}}{=} \exists e' \in \mathbf{Exit}, e' = s'.\texttt{exits}(k) \wedge e'.\texttt{key} = k \wedge is\text{-}cached_{s'}(o.\texttt{site}, e'.\texttt{siteof}) \wedge ultimate(e'.\texttt{classof}) = o.\texttt{class} \wedge ultimate(e') = o \wedge is\text{-}externalized_{o.\texttt{site}}(o,k)$$

$$ultimate(x) = \begin{cases} ultimate(x.\texttt{siteof.exported}(x.\texttt{key})) & \text{if } x \in \mathbf{Exit} \\ ultimate(x.\texttt{object}) & \text{if } x \in \mathbf{Entry} \\ x & \text{if } x \in \mathbf{Object} \end{cases}$$

$cached_{s'}(o) \stackrel{\text{def}}{=} \exists o' \in \mathbf{Object}, is\text{-}cached_{s'}(o,o')$

$$is\text{-}cached_{s'}(o,o') \stackrel{\text{def}}{=} \exists e' \in \mathbf{Exit}, e'.\texttt{replica} = o' \wedge o'.\texttt{proxy} = e' \wedge similar(o,o') \wedge present_{s'}(o.\texttt{class}) \wedge is\text{-}remote_{s'}(o, e'.\texttt{key})$$

$$similar(o,o') \stackrel{\text{def}}{=} similar(o.\texttt{class}, o'.\texttt{class}) \wedge similar(o.\texttt{site}, o'.\texttt{site}) \wedge (\forall f \in o.\texttt{class.field}, indexed(f) \Rightarrow o.\texttt{f.length} = o'.\texttt{f.length})$$

$present_s(o) \stackrel{\text{def}}{=} local_s(o) \vee cached_s(o)$

$\forall e' \in \mathbf{Exit}, re\text{-}exported_{s'}(e') \stackrel{\text{def}}{=} \exists o \in \mathbf{Object}, is\text{-}remote_{s'}(o, e'.\texttt{key}) \wedge s'.\texttt{exported}(e'.\texttt{key}) = e'$

$exported_s(o) \stackrel{\text{def}}{=} externalized_s(o) \vee (\exists k, is\text{-}remote_s(o,k) \wedge re\text{-}exported_s(s.\texttt{exported}(s.\texttt{exits}(k))))$

Table 2: Main representational properties. The *ultimate* function dereferences remote pointers until finding an object the user may see.

An object may be *present* or not. If present, the object can only be *(i)* local or, *(ii)* remote but *cached* i.e., locally represented by a replica. An object is local if the current site owns it i.e., created it (in absence of migration). A remote object has an exit item as proxy. A remote object may be re-exported even if not present.

The DMeroon representation ensures that the user can only see objects that are present (real objects or replicas). The replica and the object do have same class and same owning site. The proxy mechanism is entirely hidden. For any present object, its structural information is always present: the class, the site, the lengths of indexed fields may be accessed without any delay.

To achieve these properties, exporting an object imposes its class and its site to be exported as well. To *export* a local object is to externalize it; to export a remote object is to re-export its proxy.

The representational invariants of Table 2 are complex and mutually recursive. They must be enforced by the marshaling language and the upper protocols as will be seen in the next sections.

1.5 Summary

This Section described the basic representation of objects, proxies and tables that manage them. Associated to these representations is a set of invariants they must respect. Close to the user are the three actions that may be performed on objects. Between actions and representations are the communication channels i.e., the *object pipe* through which objects are marshaled.

2 The Marshaling Language

Sites are connected via an object pipe that conveys messages i.e., programs, conforming to the marshaling language. This Section presents the basic primitives of this very specific language that deals with distributed objects in order to allocate them, manage their replica, update them etc.

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

Table 3: Basic primitives of the marshaling language. Underlined arguments require them to be present on the unmarshaling site.

The marshaling language is expression-oriented. Its most primitive commands appear in Table 3. 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 3), it is simple to determine the number of its indexed fields and to expect as many sizes, of the proper unsigned C type, as 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 is run the `allocate` primitive.

The `fill` primitive expects an object; this object must be present. The class of the object is used to consume the subsequent bytes until the object is entirely filled. Non-pointer fields are encoded similarly to XDR or CDR [Sun89, Sie95]. 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.

This primitive relies upon the following *network invariant* which is a consequence of the invariants of Table 2: no object can be unmarshaled if its class is not present. There is a simple way to respect the previous network invariant (another better way 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 has to pull the class before pulling the object (of course, the user does not know that, the user simply asked for the object at the end of a pointer).

A site object is created via the `site` primitive. This primitive expects host and port numbers. If no such object exists in the sites table, one is created and inserted. The appropriate site is the value of this primitive. No communication is required with the mentioned site.

Remote references (exit items) 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 absent and only remotely referenced. If not yet present, an exit item is created and inserted into the exits table. As for the value of this primitive, the exit item is implicitly dereferenced if it is associated with a replica otherwise the exit item is returned as it is.

The `bind` primitive normally expects an object and an exit item; it makes the object become the replica of the exit item. Since exit items are implicitly dereferenced by the `remote` primitive, the second argument of `bind` may be a replica to mean the associated exit item. The value of that primitive is the replica associated to the second argument. If the exit item is already bound to a replica, the first argument is then ignored.

Most of these primitives are unsafe or dangerous. They all err when an argument that should be present is not. The `fill` primitive allows to overwrite objects, the `remote` primitive may confer an inappropriate class to a remote object, the `bind` primitive may associate unrelated objects and exit items. It is up to the marshaler and the upper layers (see Table 11) to ensure safety.

3 A Simplistic Marshaler

The marshaler translates values into expressions of the marshaling language. However, the compilation is not so obvious since:

1. to reduce the number of messages, objects should be pre-sent (i.e., pushed) but not too much: the compilation ought to stop.

2. the compiler must not loop while marshaling cyclic data,

3. last, the representation properties and the network invariant must be enforced.

A simplistic compiler, $\mathcal{C}$, appears in Table 4. The compiler is divided into five different sub-compilers with various properties. Predefined (and among them ubiquitous) objects are marshaled with a specific compiler hidden under the name $\mathcal{C}_{\text{predef}}$. This compiler encodes a predefined object with a `predefined` marshaling command followed by the appropriate index in the ubiquitous table of predefined objects. Sites are specially marshaled with the `site` command.

$\mathcal{C}(o : \textbf{Object}) \mapsto \mathcal{C}_{\text{presend}}(o)$
$\mathcal{C}_{\text{presend}}(o : \textbf{Object}) \wedge \textit{predefined}(o) \mapsto \mathcal{C}_{\text{predef}}(o)$
$\mathcal{C}_{\text{presend}}(o : \textbf{Object}) \mapsto \mathcal{C}_{\text{share}}(o)$
$\mathcal{C}_{\text{share}}(e' : \textbf{Exit}) \mapsto$ `remote` e'`.key` $\mathcal{C}_{\text{transmit}}(e'$`.class`$)$ $\mathcal{C}_{\text{predef}}(e'$`.site`$)$
$\mathcal{C}_{\text{share}}(o : \textbf{Object}) \wedge o$`.proxy` $\in \textbf{Exit} \mapsto \mathcal{C}_{\text{share}}(o$`.proxy`$)$
$\mathcal{C}_{\text{share}}(o : \textbf{Object}) \mapsto$ `remote` $\textit{externalize}(o)$`.key` $\mathcal{C}_{\text{presend}}(o$`.class`$)$ $\mathcal{C}_{\text{predef}}(\textit{currentSite})$
$\mathcal{C}_{\text{transmit}}(e : \textbf{Exit}) \mapsto \mathcal{C}_{\text{share}}(e)$
$\mathcal{C}_{\text{transmit}}(o : \textbf{Object}) \mapsto \mathcal{C}_{\text{presend}}(o)$
$\mathcal{C}_{\text{copy}}(o : \textbf{Object}) \mapsto$ `fill allocate` $\mathcal{C}_{\text{share}}(o$`.class`$)$ <sizes><content of o with $\mathcal{C}_{\text{transmit}}$ on pointer>
$\mathcal{C}_{\text{predef}}(s : \textbf{Site}) \mapsto$ `site` s`.IP` s`.port`
$\mathcal{C}_{\text{predef}}(o : \textbf{UbiquitousObject}) \mapsto$ `predefined` <appropriate index for o>
$\textit{externalize}(o : \textbf{Object}) \wedge o$`.proxy` $\in \textbf{Exit} \mapsto \textit{externalize}(e$`.proxy`$)$
$\textit{externalize}(o : \textbf{Object}) \wedge o$`.proxy` $\in \textbf{Entry} \mapsto o$`.proxy`
$\textit{externalize}(o : \textbf{Object}) \mapsto$ LET $k =$ new key, $e =$ new $\textbf{Entry}(k, o)$, $\textit{currentSite}$`.exported++`$[k \to e]$ IN e
$\textit{externalize}(e' : \textbf{Exit}) \wedge e' = \textit{currentSite}$`.exported`$(e'$`.key`$) \mapsto e'$
$\textit{externalize}(e' : \textbf{Exit}) \mapsto$ LET $\textit{currentSite}$`.exported++`$[e'$`.key` $\to e']$ IN e'

Table 4: Simple compiler. The *externalize* function represents the similarly named function described in Section 1.3. It returns the appropriate proxy and it extends the `exported` table (with a `++` notation) as required.

The $\mathcal{C}_{\text{copy}}$ compiler is the bulk of the whole $\mathcal{C}$ compiler, it simply encodes the content of an object. Non pointer fields are encoded similarly to XDR while pointers are compiled using $\mathcal{C}_{\text{transmit}}$. This latest compiler tries to share or pre-send objects that is, it simply chooses a compiler among $\mathcal{C}_{\text{share}}$ or $\mathcal{C}_{\text{presend}}$. The choice is currently simplistic: predefined objects are transmitted, others are shared. The $\mathcal{C}_{\text{share}}$ externalizes objects, re-exports exit items and generates a `remote` command.

This compiler is simplistic but satisfies the afore-mentioned constraints.

4 Extensions

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.

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 5. 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 communication channel. It has one occurrence 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 unmarshaled.

These operations are useful to marshal values with short cycles. This is the case for instance for **Class** and **Field** instances since a class refers to its fields that, in return, refer to the class that introduces them. To encode a class and its accompanying fields may thus be done by, first, pushing the class, so the pre-sent fields may refer to it by a `top` command and, finally, `pop`-ping the class. To mention an object with the `top`

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

Table 5: Stack marshaling commands

or `pop` commands is done with a single byte. This is to compare to a `remote` command that costs dozens of bytes.

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. It is straightforward to enrich the marshaling language with a new set of commands to manage this cache, see Table 6. Observe that a cache is specific to a one-way communication channel. 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 unmarshaled.

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

Table 6: Cache marshaling commands

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, adaptative caching policies.

In order to reduce the size of messages, DMeroon also uses the following commands, see Table 7. They don't introduce new concepts, they are pure but very common abbreviations that drastically reduce the size of messages. However they have an impact since their use makes messages non portable since the result of the marshaler depends on the sites that are at the ends of a communication channel.

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 7: Abbreviation marshaling commands

Besides the previous commands, DMeroon adds three more technical commands, see Table 8.

The two first commands, `prog1` and `prog2`[5], allow to gather expressions for their side-effects (stack or cache commands are clearly candidates). The most interesting command is the third of Table 8. The `try` command tries to unmarshal its first object (appearing as second argument). If unmarshaling this first object is free of errors then `try` acts similarly to `prog1` and returns this first object. If an error occurs while unmarshaling the first object, then `try` skips it, acts similarly to `prog2` and returns the second object (appearing as its third argument). It is always possible to skip the first object, wherever is the unmarshaling error, since its length is available in the first argument of `try`.

The `try` command is powerful since it confines unmarshaling anomalies, it also allows new strategies that respect the network invariant: no object can be unmarshaled if its class is not present. It is possible to try to send an object assuming that its class is known from the receiving site but to let the receiving site back

[5]Good old Lisp names.

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 8: Ancillary marshaling commands

up with a remote pointer if this was not the case. In other words, one may send an object with the $\mathcal{C}_{\text{try}}$ compiler defined as:

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

If the content of the object o cannot be unmarshaled 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 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.

5 Other Marshalers

The marshaling language is the target of marshalers. Often, there is more than a single way to encode values and the simplistic compiler of Table 4 may easily be extended to use more elaborated strategies with help of the previous commands. This Section investigates some of these improvements.

A marshaler may check whether the class of the object it wants to marshal is *(i)* surely known by the receiving site, *(ii)* surely unknown or, *(iii)* else. It may then choose dynamically an appropriate compiler among $\mathcal{C}_{\text{copy}}$, $\mathcal{C}_{\text{share}}$ or $\mathcal{C}_{\text{try}}$. For that inquiry, the marshaler may consult the owning site of the class or, the exits table or, the cache or stack associated with the one-way communication channels leading to or coming from the receiving site.

Trying to improve a marshaler requires some caution. For instance, to pre-send an object whose class is predefined does not seem problematic and may be expressed by the following additional rule:

$\mathcal{C}_{\text{presend}}(o : \textbf{Object}) \wedge \textit{predefined}(o\texttt{.class}) \mapsto$
  `bind` $\mathcal{C}_{\text{copy}}(o)$ $\mathcal{C}_{\text{share}}(o)$

This rule presents two problems: a termination problem and a coherence problem. The termination problem comes from the use of $\mathcal{C}_{\text{copy}}$ which uses $\mathcal{C}_{\text{transmit}}$ on every internal pointer which in turn may invoke $\mathcal{C}_{\text{presend}}$ closing the circle fatal for cyclic values. This problem may be solved with a counter, turning $\mathcal{C}_{\text{copy}}$ into $\mathcal{C}_{\text{share}}$ when deeper than a given level. This must be accompanied by a change of the architecture of the compilers since they would have to be written with a counter-passing-style.

To eagerly pre-send the content of the object may lead to multiple contents being transmitted whereas only one of them would become the replica of the remote exit item. Given that the stack and the cache may be used to retain some of these contents, the marshaler has to take care of the unmarshaler rebuilding a congruent value. We solve this problem by introducing a new layer of communication: sites will exchange requests and answers. Processing these requests/answers will provide a disciplined and safe use of the marshaling language.

5.1 Requests

A communication channel from a site to another is an *object pipe* through which are exchanged objects. The object pipe is an interesting layer since objects may be handled on the basis of their class i.e., with

an appropriate method in object-oriented parlance. The object pipe also isolates higher level needs such as coherency from low level details concerning the marshaling process.

Sites communicate through requests (and answers when needed). Requests isolates higher-level languages from the direct use of the marshaling language and provide a disciplined use of it. Coherency and other higher-level goals may be cleanly grafted into these requests or ensured with new additional requests.

OSR	**OFR**	**OFA**
object	exit	request object

Table 9: Request/Answer classes

There are two types of requests that interest us: **OSR** (standing for Object Send Request) allows to send an object (or at least a reference onto it) to another site. Conversely, **OFR** (standing for Object Fetch Request) asks the owning site of the target of a remote reference to reply with a replica by means of an **OFA** (for Object Fetch Answer; an OFA refers back to the request it answers). These are the "push" and "pull" operations for objects.

Coherency, synchronization may be achieved with **OFA** answers that update replicas according to some protocol.

Requests are structured values containing pointers, they may be considered as objects, therefore, to save code, they may be sent to other sites as objects with exactly the same machinery. However, they must be specially marshaled. The content of a request must obviously be transmitted otherwise the receiving site would only get a remote pointer instead of the request object itself. The receiving site would even not be able to ask for its content since it may only send requests that would not be transmitted for the same reason.

To solve this problem we may extend the simplistic compiler of Table 4 with the new rules of Table 10.

$$\mathcal{C}(o : \mathbf{OSR}) \mapsto \texttt{fill allocate } \mathcal{C}_{\text{predef}}(\mathbf{OSR}) \; \mathcal{C}_{\text{transmit}}(o.\texttt{object})$$
$$\mathcal{C}(o : \mathbf{OFA}) \mapsto \texttt{fill allocate } \mathcal{C}_{\text{predef}}(\mathbf{OFA}) \; \mathcal{C}_{\text{share}}(o.\texttt{request}) \; \mathcal{C}_{\text{copy}}(o.\texttt{object})$$
$$\mathcal{C}(o : \mathbf{OFR}) \wedge \textit{present}(o.\texttt{exit.class}) \mapsto \texttt{fill allocate } \mathcal{C}_{\text{predef}}(\mathbf{OFR}) \; \mathcal{C}_{\text{share}}(o.\texttt{exit})$$

Table 10: Compilation of Requests/answers.

These rules distinguish the cases for **OSR**, **OFR** and **OFA** instances. When marshaled, an **OFA** must copy the replica of its `.object` field otherwise there is no means to ensure that objects are really transmitted. This is safe since an **OFA** is the answer to an **OFR** and since an **OFR** is emitted only if the class of the expected object is already present on the emitting site.

In fact, we may eliminate the special cases of **OFR**, **OFA** and **OSR** and define a property on classes that will be taken into account by marshalers. If an object has a class with the *transmittable* property, then it is marshaled with $\mathcal{C}_{\text{copy}}$. Within $\mathcal{C}_{\text{copy}}$, a pointer field with the *transmittable* property forces the pointed object to be marshaled with $\mathcal{C}_{\text{copy}}$ instead of $\mathcal{C}_{\text{transmit}}$. The **OFR**, **OFA** and **OSR** classes are transmittable; the `.object` field of **OFA** is also transmittable. Class definitions may customize the marshaler but ought to ensure the good behavior of the enriched marshaler.

New requests may be invented to reduce the size of messages. For instance, the **ORR** (standing for Object Refresh Request) asks for the content of an already present replica. Only the mutable fields have to be transmitted back. **ORR** requests may be implemented with a new appropriate compiler, $\mathcal{C}_{\text{refresh}}$, emitting a new (unsafe) `refresh` marshaling command whose use is wrapped within an **ORA** (for Object Refresh Answer).

5.2 Behaviors

It is now clear that the marshaler may evolve at run-time provided it still generates appropriate expressions of the marshaling language. The marshaler may then, at run-time, copes with memory exhaustion (and thus caches less objects or even resets caches), reacts to a reduction of network bandwidth (and thus refrains to pre-send objects or increase caching), deals with specific (cyclic for instance) values (and lets the user provide an appropriate marshaling technique for these classes).

If the marshaler is itself reified into an object, it may be transmitted on a site and substituted to the former one. Marshaling may thus be dynamically improved using marshaling. The reification of the marshaler may look like a byte-code vector or as a graph of nodes corresponding to compilers linked with edges checking for some conditions of use.

6 Related Work

We will only discuss the aspects that are related to the marshaling language and ignore the distributed aspects of DMeroon implementation with respect to other systems such as [Ach93]. The first Section of this paper are only a support for the exposition of the marshaling language.

The XDR (for eXternal Data Representation [Sun89]) 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 unmarshaler 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 from a `.h`-like file.

Corba introduced CDR (for Common Data Representation [Sie95]) for marshaling as well as seven types of request/answer in the IIOP protocol (for Internet Inter-ORB Protocol). Since Corba brings the notion of object identity, references to remote objects may easily be 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).

Static generation of marshalers/unmarshalers produces a code whose size may be extremely large. Some alternate solutions were explored. [**?**] proposed to interpret type descriptors and showed that the speed is not too much deteriorated since the unmarshaler is very compact and fits well in processor cache. [Bar97] also proposed a kind of interpreter that he called *the marshaling engine*. Finally, [Hof97] lessens the need for space with just-in-time stub generation.

Within 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 unmarshaler weights 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/unmarshaler 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 accomodated by `rpcgen`).

With respect to speed, our solution is clearly slower than XDR-style compiled code but it offers other advantages. The marshaler may 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. Our solution is portable, does not depend on the operating system and tolerates a GC recycling unused classes for instance.

7 Conclusions and Future Work

This paper proposed a layered architecture shown on Table 11.

The first part of the paper covers the terminology and the detailed representation of objects, references and replicas as well as the representational invariants they have to respect. This Section is quite subtle since

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

Table 11: Layers

this representation allows for the existence of meta-classes, for classes to be dynamically created once and instantiated everywhere.

The second part of the paper exposes a marshaling language describing the streams of bytes that allow one site to transmit structured values to other sites. The marshaler is a compiler that turns values into expressions of the marshaling language while the unmarshaler is a byte-code interpreter evaluating expressions of the marshaling language to build structured values. The language is inspired by programming languages and allows for versatility both at compile-time and run-time.

We think that the small glimpses at formalism here and there in this paper may be a hint to some proof systems:

- 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 developp 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 the net [Que98].

Many thanks to Luc Moreau for his fruitful proof-reading.

References

[Ach93] Bruno Achauer. Implementation of distributed trellis. In Oscar M Nierstrasz, editor, *ECOOP '93 — 7th European Conference on Object-Oriented Programming*, volume Lecture Notes in Computer Science 707, pages 103–117, Kaiserslautern (Germany), July 1993. Springer-Verlag.

[Bar97] A Bartoli. A novel approach to marshalling. *Software Practice and Experience*, 27(1):63–86, January 1997.

[BC87] Jean-Pierre Briot and Pierre Cointe. A uniform model for object-oriented languages using the class abstraction. In *IJCAI '87*, pages 40–43, 1987.

[Hof97] Markus Hof. Just-in-time stub generation. In *JMLC'97 — Joint Modular Languages Conference*, pages 197–206, Linz (Austria), March 1997.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *POPL '92 – Nineteenth Annual ACM symposium on Principles of Programming Languages*, pages 39–50, Albuquerque (New Mexico, USA), January 1992.

[MDF97] Luc Moreau, David DeRoure, and Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. In *Third International Europar Conference (EURO-PAR'97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 581–590, Passau, Germany, August 1997. Springer-Verlag.

[Mos90] J. Eliot B. Moss. Working with objects: To swizzle or not to swizzle? Technical Report 90–38, University of Massachusetts, Amherst, Massachusetts, May 1990.

[Piq91] José Miguel Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 – Parallel Architectures and Languages Europe*, pages 150–165. Lecture Notes in Computer Science 505, Springer-Verlag, June 1991.

[PTM95] 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.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu 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), October 1993.

[Que94] Christian Queinnec. Locality, causality and continuations. In *LFP '94 – ACM Symposium on Lisp and Functional Programming*, pages 91–102, Orlando (Florida, USA), June 1994. ACM Press.

[Que95] Christian Queinnec. DMeroon: Overview of a distributed class-based causally-coherent data model. In Takayasu Ito, Robert H Halstead, Jr, and Christian Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, Lecture Notes in Computer Science 1068, pages 297–309, Beaune (France), October 1995.

[Que97] Christian Queinnec. Sérialisation–désérialisation en DMeroon. In Omar Rafiq, editor, *NOTERE97 — Colloque international sur les NOuvelles TEchnologies de la RÉpartition*, pages 333–346, Pau (France), November 1997. Éditions TASC.

[Que98] Christian Queinnec. DMeroon *A Distributed Class-based Causally-Coherent Data Model – General documentation*. LIP6, 1998. See also Rapport LIP6 1998/039 ¡http://www.lip6.fr/reports/lip6.1998.039.html¿.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), nov 1992. Also available as Broadcast Technical Report #1.

[Sie95] Jon Siegel. *Corba, Fundamentals and Programming*. John Wiley and Sons, 1995.

[Sun89] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March 1989.