DMeroon

A Distributed Class-based Causally-Coherent Data Model

– General documentation – Revision: 1.64 –

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

DMeroon provides a data model above a coherently distributed shared memory. DMeroon allows multiple users to statically or dynamically create new classes hierarchically organized, to dynamically instantiate these classes and, to dynamically and coherently share the resulting instances over a network. DMeroon automatically takes care of representation and alignment, marshaling and unmarshaling objects, migrating and sharing objects, local and global garbage collections.

This document describes DMeroon, its philosophy of design, its architecture and principles of implementation, and its bindings with various languages. It also presents some internal details within DMeroon or some applications above DMeroon.

This document tries to present the overlines of DMeroon*, in places, it describes features which are not implemented, in some other places there are implemented features that are not documented. I packed it up in order for interested people to get an idea and, perhaps, induce them to pursue my effort or definitively convince me of its little value. I have a lot of lectures to prepare for the following months and will not be able to devote much time to* DMeroon*.*

Contents

List of Figures

Chapter 1

Introduction to DMeroon

DMeroon was conceived with many different goals in mind: portability with respect to hardware and languages, expressiveness, dynamicity, coherency, reusability, self-description etc. DMeroon is the data layer of a distributed language, named Icslas, which is still under progress. Nevertheless DMeroon has been carefully designed so it can also be used, independently of Icslas, for various purposes:

- as a class-based system,
- as a marshaling engine,
- as a message passing layer,
- as a distributed shared memory.

DMeroon is based on a number of research results and offers unique features:

1. language design [Que93],

   DMeroon offers a class-based system with regular or indexed fields without inheritance restriction. Objects can be shared if remotely referenced but still retain their identity.

2. object-oriented self-description [QC88, Que90].

   Objects are described by their classes and classes are objects as well. Most of the implementation of DMeroon can be freely inspected programmatically. The object system is distributed and supports multi-users working together i.e., dynamically creating new classes and dynamically exchanging instances of them.

3. shared data coherency [Que94a, Que94b],

   Mutation of remotely shared mutable objects is coherently i.e., causally propagated to sites caching copies of these objects through a lazy invalidation protocol. DMeroon supports the notion of remote pointer i.e., a field of type `reference` may refer to any kind of DMeroon object independently of its location.

4. marshaling engine [Que97b],

   Marshaling objects is considered as a compilation process, the resulting stream of bytes is then considered as a program which is interpreted by a deserialization interpreter. The exchange protocol is therefore specified as a language and, most often, many different strategies of compilation exist; for all of them the unmarshaling interpreter remains constant.

5. garbage collection design [LQP92],

   Remotely shared data are managed with a reference counter. Useless objects or classes automatically disappear, cycles of garbage are also recycled via another GC technique.

DMeroon may be used for very different purposes, see figure 1.1, as well with various languages for which a binding exists (Scheme (Bigloo, Scheme→C, OScheme, SCM, ...), C, Emacs, Tcl, all under progress).

- First, you may use DMeroon, stand alone, in a single process i.e., as a single site, just to take benefit of its library of classes and objects. You may define your own classes, instantiate them, read or modify fields within these objects. You may also describe or access C structured values with DMeroon classes. Garbage collection is active if possible. You may inspect or modify DMeroon objects via the HTTP protocol and thus control the parameters of your application in the spirit of Paws[1]. You may also extend the DMeroon server to support additional protocols.
- Second, you can start multiple sites (i.e., processes on possibly different machines) and let DMeroon be the marshaling/unmarshaling engine. You then copy structured values from site to site. Regular unstructured values such as messages i.e., sequences of bytes, may also be exchanged if represented by instances of `String`.
- Third, your application may be supported by the shared distributed memory provided by DMeroon, you may then use shared objects. Examples of such applications are the distributed management of DMeroon sources, distributing computations (farming) upon a synchronizing memory, archiving structured and complex data for a variety of processes.
- Fourth, DMeroon may be the unifying framework that hosts, on a variety of sites, a variety of applications allowing you and your colleagues to cooperate above a coherent memory.

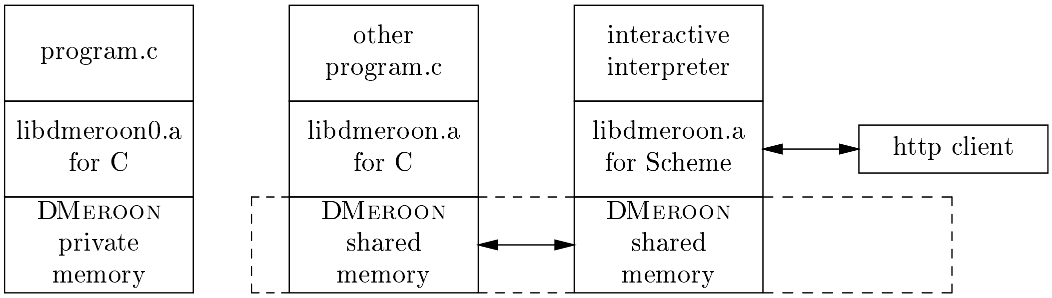

Figure 1.1: DMeroon overview

1.1 About this document

This document is a *pot-pourri* around DMeroon. It first describes the generic aspects of DMeroon then details its current bindings (with C and Scheme). Additional appendices describe some finer points of the implementation (as students keep asking me!).

> **Implementation note:** *Warning! This document is under construction as recalled in some sections. I also sometimes comment finer points inside implementation notes such as this one.*

1.2 About DMeroon

DMeroon is freely accessible under Gnu Library General Public License version 2. All remarks, bugs, patches, extensions, stories etc. are to be mailed to:

`Christian.Queinnec@lip6.fr`

Online fresh information is available from:

[1] http://zenon.inria.fr:8003/koala/phk/k-web/intro.html

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

Additional questions may be addressed to:

DMeroon@spi.lip6.fr

To subscribe to this unmoderated mailing list, send a mail to:

DMeroon-request@spi.lip6.fr

Insults are left to the following mailing list if one cares to create it:

DMeroon-haters@to.appear.somewhere

Finally, an ever-changing DMeroon server is sometimes reachable at:

http://youpou.lip6.fr:56423/

1.3 Frequently Asked Questions

<< Section under construction >>

◇ *what means* DMeroon*?*

The name DMeroon stands for "Distributed Meroon" where Meroon is the name of an object system for Scheme. Informations on Meroon is available on:

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

Meroon's own FAQ tells us that the name Meroon was the name of my son's teddy bear.

◇ *How about PVM?*

DMeroon is different from PVM for at least two reasons: DMeroon provides you a distributed shared memory and structured objects with pointers. Conversely PVM offers task control, host management whereas DMeroon only deals with data. Due to its youth, it is not ported as PVM is.

◇ *How about Corba or ILU?*

DMeroon is different from Corba and ILU: it does not provide multiple inheritance of interfaces nor remote method invocation. It offers you automatic coherency for shared mutable objects, source code and extensibility due to the self-description of the implementation. However, it might be interesting to let DMeroon understands the IIOP protocol. Corba offers pointers to remote interfaces whereas DMeroon offers real mobile objects.

◇ *How about Java?*

DMeroon is different from Java: it offers you indexed fields allowing multiple values to be packed together (so they are not split apart when marshaled), it offers a distributed shared memory and automatic coherency. With Java RMI, a remote object is represented by a stub implementing some interface, the class of the local stub and the class of the remote object are not comparable whereas in DMeroon they are the same. Moreover in Java, the stub and the server object are compelled to inherit from special classes, this is needless in DMeroon. DMeroon also offers handling C values (unsigned int for instance) with C semantics.

1.4 Acknowledgments

Jing Wang wrote the first version of the http protocol part that allows to modify mutable fields of objects. Jean-Michel Inglebert contributed with `dmClient` a tool to write and debug DMeroonScript programs.

Chapter 2

Fundamental data model

A DMeroon object is an ordered contiguous sequence of fields, see figure 2.1. Fields may be regular (mono-field) or indexed (poly-field). A regular field holds a single value while an indexed field holds an ordered sequence of values whose number of elements is determined at instance creation time (currently this number is stored right before these values). This uniform model allows to represent all mundane data structures such as records, vectors, strings with full inheritance ability[1].

The object shown on figure 2.1 starts with two regular fields, followed by an indexed field, itself followed by a regular field. As in C, a DMeroon reference refers the first byte of its content. DMeroon also stores a reference to the class of the object as well as a reference to its proxy (a DMeroon object containing all the information needed to manage distribution) in a prefixed header which itself may be prefixed by the standard header required by the binding language if any (see figure A.2, page 80).

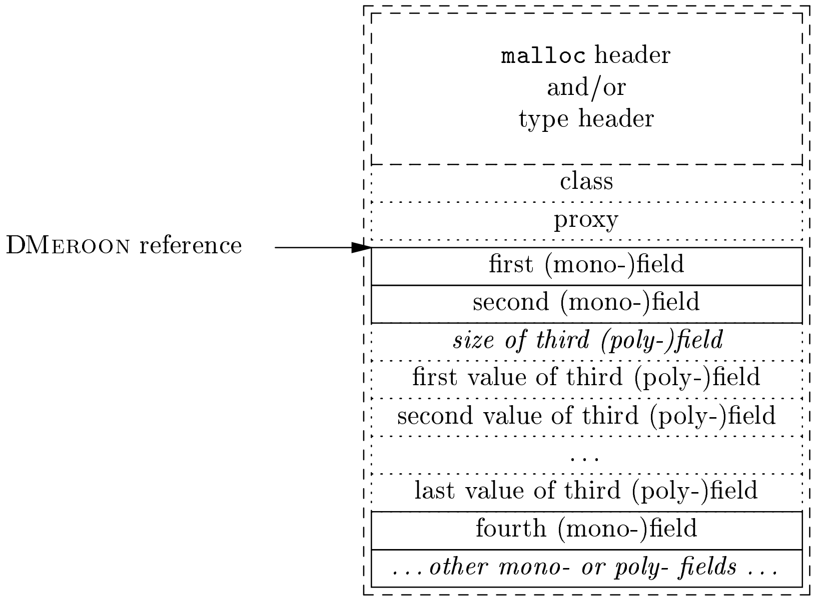

Figure 2.1: DMeroon instance example

[1] Never dreamt of a colored string ? That is, from an implementation point of view: a repetition of characters followed by a color.

2.1 Types of fields

All fields have a precise type as shown in table 2.1. These types have sizes and alignment constraints that depend on the site (the hardware running DMeroon) and on the C compiler used to compile the DMeroon library. The sizes mentioned in the table are the usual ones. Every implementation of DMeroon must implement all these basic types with their associated range but, on a particular site, some fields may be bigger than indicated to implement the correct range of values.

> **Implementation note:** *For any machine I know, these sizes are the real sizes.*

Every DMeroon type has an associated class with a single field, the `value` field, of that type. Hence `nat1` is associated to class `Nat1`, etc. These classes are used in the implementation of DMeroon and may be used or refined by interested users, see table 2.3.

For every C structure that has an equivalent DMeroon class, the layout of the structure and the layout of the class are the same (same order of fields, same sizes, same alignment constraints). The DMeroon model is richer since it allows indexed fields, it is also poorer since C offers bounded arrays that are currently not part of the DMeroon model.

> **Implementation note:** *C structures are subsumed by* DMeroon *classes at the exception of arrays. For instance,* `char s[5]` *is currently not supported in* DMeroon*, you must use instead an indexed field or five contiguous regular fields.*

Some basic types deserve special explanation.

- Signed integer types have a symmetric range to be free of one-complement or two-complement encodings.

  > **Implementation note:** *XDR then Corba adhere to two-complement, I may change on that point.*

- The `reference` type is used to hold a pointer to another DMeroon instance. The referenced instance can be local to the current site or remote without any semantical differences.

  > **Implementation note:** *A reference to a* DMeroon *instance is the address of the first byte of its first field. This makes easy to allocate C structures with* DMeroon *allocation functions and to handle them as C or* DMeroon *objects. For languages in which objects are described by the address of the first word of their header (Bigloo for instance), pointers have to be translated back and forth between* DMeroon *and the binding language. See figure A.2 for some implementation schemes.*

- The `data` type can be used to record any non-DMeroon information that fits in the size of a pointer to a local C datum i.e., a `void*` value. Such local information can be a file descriptor, a Scheme value, an EBCDIC character, etc. You may transmit such a field, without DMeroon blessing, that is, DMeroon does not ensure you anything on the size or on the representation of the transmitted field. However if you use homogeneous machines, you may try.

- The `code` type is similar but may hold the address of a C function. The same lack of DMeroon blessing holds of course. You may try it if you are sure that your functions are linked with similar addresses on different machines.

- The "`netnat`*" types are similar to their "`nat`*" counterpart except that these natural numbers are stored in "network order". Some data structures, generally related to network structures (`sockaddr_in` for instance), impose this encoding, it is therefore useful and less painful to let DMeroon manages this encoding for you. That is, you never have to use "`netnat`*" numbers only their equivalent "`nat`*" numbers.

index	**name**	**size (bytes)**	**meaning**
0	`nought`	0	not instantiable
1	`nat1`	1	natural (unsigned) number from 0 to $256^1 - 1$
2	`nat2`	2	natural (unsigned) number from 0 to $256^2 - 1$
3	`nat3`	4	natural (unsigned) number from 0 to $256^4 - 1$
4	`nat4`	8	natural (unsigned) number from 0 to $256^8 - 1$
5	`reference`	4 or 8	pointer to a DMeroon instance
6	`data`	4 or 8	non-DMeroon information (the size of a `void*` C variable)
7	`code`	4 or 8	address of a C function
8	`int1`	1	(signed) small integer from $-256^1/2 + 1$ to $256^1/2 - 1$
9	`int2`	2	(signed) small integer from $-256^2/2 + 1$ to $256^2/2 - 1$
10	`int3`	4	(signed) small integer from $-256^4/2 + 1$ to $256^4/2 - 1$
11	`int4`	8	(signed) small integer from $-256^8/2 + 1$ to $256^8/2 - 1$
12	`char1`	1	ISO Latin1 character
13	`netnat2`	2	natural (unsigned) number from 0 to $256^2 - 1$ in network order
14	`netnat3`	4	natural (unsigned) number from 0 to $256^4 - 1$ in network order
15	`unicode`	2	*not yet implemented*
16	`float1`	4	IEEE floating point number with 32 bits
17	`float2`	8	IEEE floating point number with 64 bits
18	`float3`	16	IEEE floating point number with 128 bits
19	`offset`	4	*internally needed by* DMeroon
20	`date`	8	date following Unix convention
21	`key`	16	*internally needed by* DMeroon
22	`class-options`	4	*internally needed by* DMeroon
23	`field-options`	4	*internally needed by* DMeroon
24	`port-options`	4	*internally needed by* DMeroon
25	`site-options`	4	*internally needed by* DMeroon
26	`external-options`	4	*internally needed by* DMeroon
27	`context-options`	4	*internally needed by* DMeroon
28–63			*reserved by* DMeroononeone

Table 2.1: DMeroon primitive types

- The "`*4`" types are not necessarily usable everywhere since they often occupy eight bytes and not all hardware are able to add or divide 8bytes entities. It is nevertheless possible to migrate objects with such fields from site to site, but you may not have, in some language binding, the possibility to manage (i.e., read or write) these 8bytes entities. Nevertheless these types exist and are predefined.

- The `float3` type is not implemented everywhere due to the lack of a common representation. It was initially intended for `long double` of C but, unfortunately, their size may be 8 (sparc-bsd), 10 (ieee 80bits), 12 (linux) or 16 (sparc-solaris) bytes.

  Implementation note: *Types are encoded with a single byte. This leaves around 200 types free for future extension. If you plan to introduce new statically predefined types, remember that you will not be able to communicate with servers lacking them. However if your additional types are valuable, make others share them! Future releases of* DMEROON *will probably use more predefined types for internal usage that is why some bytes are reserved.*

- The `key` is an internal DMEROON type used to name objects that are to be known via the network. A key qualifies a single object and is never reused. It is difficult to forge meaningful keys *ex nihilo*, this allows to protect hidden remote objects from being discovered.

  Implementation note: *Currently, a key is composed of the IP number of the* DMEROON *server where it was allocated, of the portnumber used by this server, of the date of creation, of the value of a 16bits cyclic counter and, finally, of 32bits randomly computed. I may adopt the DCE naming scheme some day.*

2.2 Access to fields

Any DMEROON instance can deliver the content of its regular fields. Given an additional index i, it can also return the content of the i-th value of an indexed field. Of course, it is not possible to access a field that does not exist in an instance and, for indexed fields, indexes are checked to be within the range defined by the instance. Mutable fields can be altered with the same limitations. Indexed fields can return the number of values they hold on a per-instance basis.

The precise way to access, inquire or modify a field in a DMEROON instance depends on which binding language is used with DMEROON: see, for instance, the C binding (chapter 3) or the Scheme binding (chapter 4). However access should always be performed via the appropriate functions to ensure the consistency and the safety of DMEROON space and to give time to DMEROON to serve http requests or to manage, i.e., garbage collect, the memory.

A field is qualified by some options that conditionalize the access, see table 2.2. These options may be freely combined. By default, fields are not mutable, not volatile, not local and not secret.

kind	**meaning**
mutable	The field can be modified through the API.
volatile	The content of the field may evolve spontaneously.
local	Never marshal the content of this field, its meaning is only local.
secret	The field cannot be read.

Table 2.2: DMEROON field options

DMEROON prevents the mutation of immutable fields. Only mutable fields may be written. By default, fields are immutable.

A volatile field cannot be cached: it must be read every time it is accessed. Even if the user cannot modify it i.e., the field is immutable, its content may evolve. Volatile fields are often used by the DMEROON implementation to allow the inspection of immutable but evolving values.

A local field cannot be read outside of the current site: it only has a local meaning. By default, fields are not local. Local fields are handy for fields of the `data` or `code` type and may also be used to keep fields secret.

A secret field cannot be read at all. It is probably not very interesting to be volatile at the same time. This option is handy for some internal fields reserved to the implementation.

A field of type `reference` is always restricted to only refer to instances (whether direct or indirect) of a given class. The root of all classes that is, `Object`, allows for all DMeroon objects.

2.3 Distributed aspects

DMeroon manages the DMeroon space which is the union of all DMeroon objects. Within a process, the user space is the complement of the DMeroon space: values of `data` and `code` fields usually refer to the user space. It is possible to perform computations in the user space then to copy results in the DMeroon space to share them or exchange them with the marshaling/unmarshaling capacity of DMeroon.

A reference to a DMeroon object from the user space always refers to an instance such as the one shown on figure 2.1. This is an invariant maintained by DMeroon: the user of the API never sees internal DMeroon objects (such as Entry or Exit items).

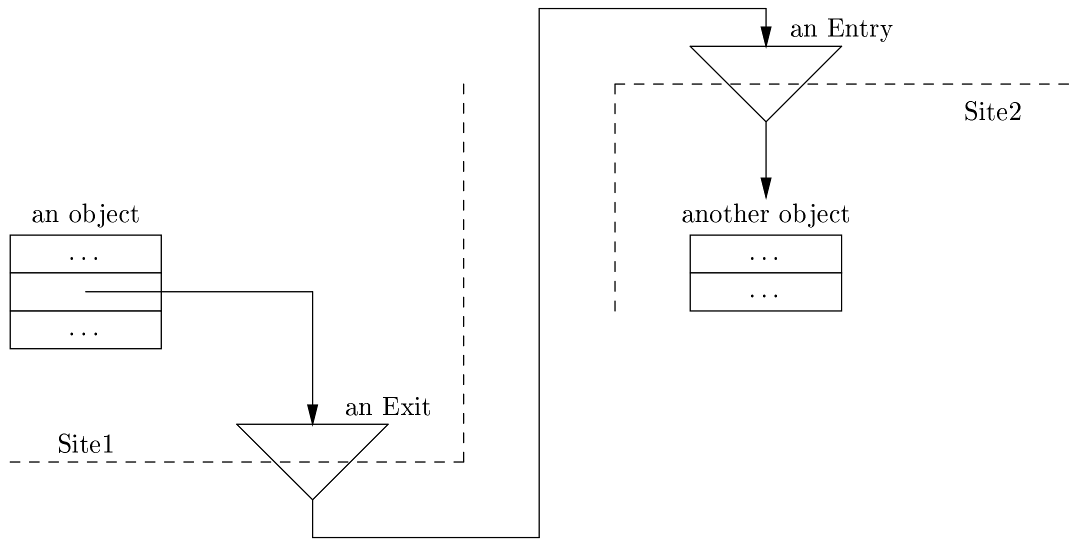

Figure 2.2: A remote pointer

The content of a field of type `reference` may be null or not. If non-null then it refers to a DMeroon object. It is possible to merge its own DMeroon space with other DMeroon spaces if they are available via the network. When merged, referenced objects may belong to one or another site without semantical differences: users do not need to be aware of distribution but for speed. If the referenced object is local, the reference is represented by a regular pointer; if the referenced object is not local, then a remote pointer is used instead as shown on figure 2.2.

When a field of type `reference` is read, the DMeroon object which is referred to is brought locally i.e., is cached and a reference onto it is returned to the user (see figure 2.3). The cached object is also known as a *replica*. A coherency protocol is run by DMeroon to ensure the consistency of the cached replica with respect to communications between sites. Access to DMeroon objects should only be done through the DMeroon API in order to maintain the consistency. This is costly with respect to direct access with offsets as in C but allows DMeroon to garbage collect its space, maintain consistency, serve requests and the like. DMeroon allows you to send references onto objects towards remote sites. Then all information exchanges between sites (if represented by objects) are noticed by DMeroon so it may ensure coherency.

Reading a field of a cached object may be efficient under certain circumstances (not volatile, validly cached). Writing a field always implies sending a message to the site that owns the original object asking

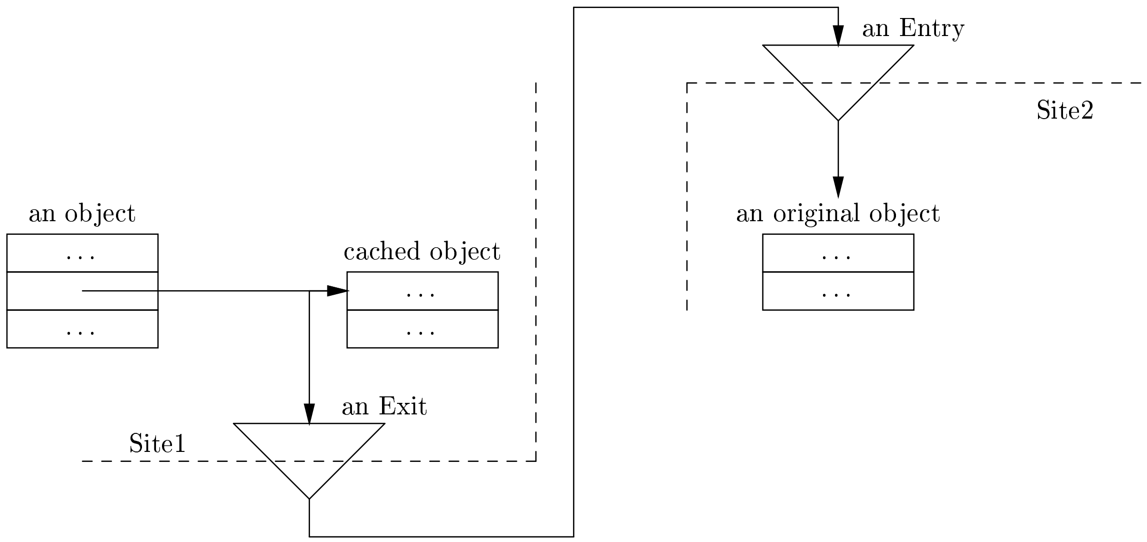

Figure 2.3: Locally cached remote object

that site to perform the mutation. These details are fortunately hidden for the users.

2.3.1 Clocks

A shared mutable object is monitored by a clock, an instance of the predefined class `Clock`. When such an object is mutated, its clock is incremented. Whenever sites exchange messages, they lazily propagate the clocks they are aware of i.e., their clocks and others.

When an object is cached on a site, its associated clock is also cached. Moreover, the exit item associated to the object memorizes the time when the cached object was read. For instance, on figure 2.4, the replica was read at time 14.

Before a replica is read, its associated clock is checked not to have changed since the replica was fetched. On figure 2.4, Site1 is not aware of any change on this clock so the replica is not obsolete. Had a message been sent from Site2 recently, then Site1 would have advanced the clock to 15 thus making obsolete the replica (see figure 2.5). When the replica is out of date, a request for a fresher copy will be sent (this will authorize Site2 to advance its (cached) clocks with respect to Site1). This is the essence of the lazy invalidation protocol described in [Que94a].

There is a trade-off between the number of clocks to propagate and the number of objects that are simultaneously invalidated because they are monitored by the same clock. DMEROON allows you to choose the right grain: you may create clocks (or even subclass the `Clock` class) and associate them to objects on a per-object basis.

All entry items have a reference counter counting the number of remote pointers referencing an object. This allows for a simple garbage collector. A more elaborated marking GC is run by DMEROON to take care of useless distributed cycles.

2.4 Sites

A site corresponds to a single address space (usually a process in Unix parlance). Sites may be connected to form a bigger DMEROON space.

When a DMEROON object is allocated on a site, it is owned by this very site. If the object is motile, it may migrate that is, be owned by another site. The site that owns an object serializes the mutations that are applied on this object. It is possible to know which site owns an object.

An object is *local* to a site if it owned by this site, otherwise the object is *remote* and associated to an Exit item i.e., remotely pointed. An object is *present* if local or cached. An object is *absent* if remote and

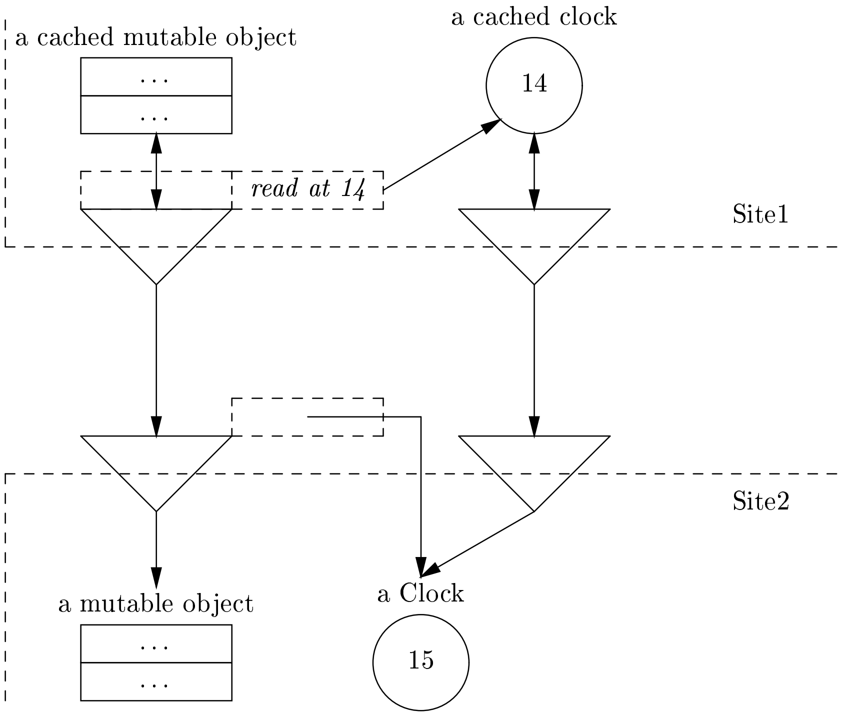

Figure 2.4: Object monitored by a clock with an up to date replica

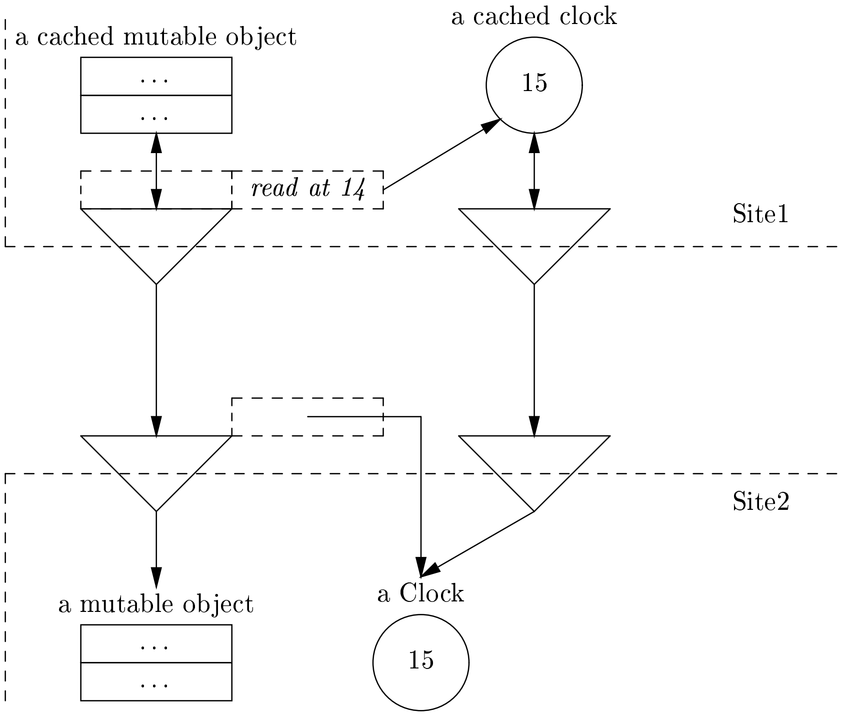

Figure 2.5: Object monitored by a clock with an obsolete replica

not cached i.e., not associated to a replica on the current site. A replica is *valid* or, a remote object is *validly cached* if its associated clock is not known to be out of date as perceived by the current site (see figure 2.5).

Sites are reified into instances of the immutable `Site` class. These objects may be read to discover the properties of the site. A site has an `information` field that holds values that are said to be published i.e., readable by any one in possession of a reference onto that site object.

2.5 Class-based model

Any DMeroon instance belongs to a particular class, the class of which it is a direct instance. Classes are related by inheritance. As MeroonV3 or Java, DMeroon is a single inheritance class-based model. When a class A inherits from a class B, all the fields of B are also fields of A with the same properties. Therefore when a class is defined only the additional fields it introduces are mentioned.

DMeroon predefines a hierarchy of classes, most of them can be subclassed at will without restriction. The actual core hierarchy is shown on figure 2.3 where indentation means "subclass of", for instance `Type` is a direct subclass of `Object`, `DMeroonType` is a subclass of `Type` and therefore an indirect subclass of `Object`. The implementation of DMeroon ensures that the *is-a-subclass* check (when comparing two classes) or the *is-an-instance* check is always performed in constant time. Furthermore, in the current implementation of DMeroon, all predefined classes are statically allocated and not dynamically created i.e., they are ubiquitous.

Classes may be customized with some options. These are:

- **non-subclassable**: it is not possible to create a subclass of such a class (but predefined sub-classes may exist).
- **immotile**: instances of that class cannot be migrated i.e., change their owning site (they may be shared i.e., remotely pointed instead).
- **non-instantiable** characterizes a class that cannot be instantiated (this is reserved for some internal classes of DMeroon).
- **mutable**: instances of such a class have at least one mutable field. This option is deduced from the options of the fields.
- **volatile**: instances of such a class have at least one volatile field. This option is deduced from the options of the fields.
- **copyable**: instances of such a class are, by default, copied rather than shared when communicated from one site to another one.
- **sharable**: instances of such a class are, by default, shared rather than copied when communicated from one site to another one.

2.5.1 Main predefined classes

Only the core classes are defined here. Many more useful classes also exist and will be described elsewhere in particular those of the Icslas language.

`Object` [*Class*]

This class is the root of the inheritance tree. All classes inherit from `Object` therefore all objects are (direct or indirect) instances of `Object`. The `Object` class has no field. The `Object` class is itself an instance of `Class`.

```
Object                   the root of all classes
  Type
    CType                types of the C support language
    DMeroonType          DMeroon portable types
  Class
  Field
  String                 repetition of char1
  MutableVector          mutable vector of DMeroon references
  ImmutableVector        immutable vector of DMeroon references
  ...
  Clock
  Hashtable
  BoundedStack
  ImmutableLink
    ImmutablePair        immutable dotted pairs à la Lisp
  QueueLink
  Symbol                 immutable symbols à la Lisp
  Boolean                Booleans as in Smalltalk
    True                 #t as in Scheme
    False                #f as in Scheme
  ImmutableByteVector
  Null                   nil as in Lisp
  ...
  Site                   the class of the current site
  Dictionary             dictionary node, see appendix C.6
  Value                  Classes for all DMeroon types as in Java
    Number
      ExactNumber
        SignedNumber
          Int1
          Int2
          Int3
          Int4
        UnsignedNumber   Classes for unsigned numbers (not as in Java)
          Nat1
          Nat2
          Nat3
          Nat4
      InexactNumber
        Float1
        Float2
        Float3           not always present
    Char1
    NetNat2              Nat2 in network-order
    NetNat3              Nat3 in network-order
    Data                 reference to non-DMeroon values
    Code                 reference to C functions
    Reference            reference to a DMeroon object
    Date                 portable date
  ...
  Context              used for bindings
  Facts                asynchronous facts reported by DMeroon
```

Table 2.3: Hierarchy of predefined classes

`Class`		*[Class]*
`options`	*a sequence of bits encoded as a* `class-option`	
`super`	*an indexed sequence of* `reference` *to the superclasses*	
`field`	*an indexed sequence of instances of* `Field`	
`name`	*a repetition of characters*	

This class describes the structure of classes. All classes are direct or indirect instances of `Class`, particularly `Object` and `Class` according to the ObjVlisp model [Coi87]: `Class` inherits from `Object`, `Object` is an instance of `Class` and so is `Class` itself.

The `options` field describes some properties of the class such as its immutability or motility etc. This field is reserved for DMeroon's usage. An instance of an immotile class cannot migrate from a site to another one: it is nailed down to the site where it was created. An immotile instance may be copied or remotely referenced hence shared. A non-subclassable class cannot be sub-classed. An immutable class imposes its immutability to all its fields.

The indexed `super` field refers in order to the super-classes. `Object` is the super-class of index 0, the last super-class is the class itself.

The indexed `field` field refers to instances of `Field` describing the fields of the class.

`Field`		*[Class]*
`options`	*a sequence of bits encoded as a* `field-options`	
`introducing`	*a* `reference` *to the class that introduced the field*	
`referring`	*a* `reference` *to the class the values should be instance of*	
`offset`	*an indexed sequence of* `offset`	
`name`	*a repetition of characters*	

This class describes how to access fields within instances. A field descriptor encapsulates all the information needed to access or introspect such a field in an instance of such a class.

The `options` field describes some properties of the field such as its immutability etc. This field is reserved for DMeroon's usage.

The `introducing` field refers to the `Class` instance that introduced that field i.e., the class from which this field is inherited.

When the field holds reference(s), the `referring` field contains the class the referred values should be (direct or indirect) instances of. This field is useless for other DMeroon types.

The indexed `offset` field (a kind of `nat3`) contains raw numbers encoding the type of the value of the field, the type of the repetition factor if any and how to access this(these) value(s).

`Type`		*[Class]*
`index`	*a* `nat1` *index characterizing this type*	
`alignment`	*a* `nat1` *number constraining the allocation*	
`size`	*a* `nat2` *size (in bytes)*	
`repsize`	*a* `nat2` *size (in bytes)*	
`name`	*a repetition of characters*	

This class describes the implementation of a DMeroon type. All DMeroon types are characterized by an `index`, a small natural number. Conversely, this index allows to retrieve the associated type. The `alignment` field expresses the constraints of the hardware (as understood by the C compiler) to record a value of that type. Values of that type have a length of `size` bytes per se and a length of `repsize` (that is `size` plus some padding) when repeated (as in an array of that type).

Two sorts of types exist: C types and DMeroon portable types, they are described by special subclasses of `Type`: the `CType` and `DMeroonType` classes. C types describe the implementation and are initialized by the C compiler used to compile DMeroon. DMeroon portable types have a field telling which C type is used to implement them. Reciprocally, C types have a field telling which DMeroon type might implement them. C types are only used by the implementation and are not to be seen by normal users.

`String` [*Class*]
  `character` *a (*`nat3`*-indexed repetition of characters*

This class describes immutable strings.

`Symbol` [*Class*]
  `character` *a (*`nat3`*-indexed repetition of characters*

This class describes symbols la Lisp also known as unique strings in Java parlance. Two different symbols differ at least by a character or the number of characters they have. Symbols are immutable.

2.6 Abstract API

This section summarizes the abstract primitives offered by DMeroon. Of course, their exact interface depends on the binding language. All details are not included here, the precise documentation should be referred to.

Primitives are sorted by level on table 2.4. Level 0 corresponds to primitives that may run on a single site, level 1 are for simple distributed primitives while level 2 corresponds to more complex primitives. Primitives of level 0 are generally immediate i.e., non blocking, while others may require communication with remote sites and therefore are non immediate.

Implementation note: *Currently, level 2 primitives are still in a flux.*

DMeroon functions or variables all have a name starting with the `DM` prefix: this prefix may be longer. Due to the binding language, dash or underscore may be additionally used. When a call is erroneous, the error is reported in a way that depends on the binding language. Function calls often return some useful data in a way that also depends on the binding language.

An *immediate* primitive is a primitive that answers in a bounded time. Primitives that require communication with other sites may take a long time hence are not immediate. It is possible, with some bindings, to bound the duration of a call to such a primitive and to return an error when this duration is exhausted without answer.

2.6.1 Initialization

Before using DMeroon, you must initialize it with `DM_initialize`. This function is sometimes implicitly called but it is harmless to call it again. It returns an object that describes the current site. By default, a DMeroon server is started that listens to a TCP port. The port number is currently defined as the value of the `DMEROON_SERVING_PORT` environment variable or, by default, as the value of the `DM_DMEROON_FAVORITE_PORT` cpp variable which is currently:

```
#define DM_DMEROON_FAVORITE_PORT  56423
```

Level	Functionality
0	initialize DMeroon library
0	check whether an instance is an instance of a class
0	return the direct class of an instance
0	get the content of a regular field within an object
0	get the content of a indexed field within an object
0	get the length of a indexed field within an object
0	set the content of a regular field within an object
0	set the content of a indexed field within an object
0	instantiate a class
0	declare the end of the initialization of an object
0	create a subclass
0	get the site that owns the object
2	set the site that owns the object (i.e., migrate the object)
2	set the clock associated to a mutable object
0	get the clock associated to a mutable object
1	connect the current site to a remote site
1	send to another site a reference onto an object
1	receive a reference
2	check whether two pointers refer to a same object

Table 2.4: DMeroon primitives

2.6.2 Class

A DMeroon instance has a single class of which it is a direct instance of. This class may be obtained with `DM_class_of`. Inversely, it is possible to check with `DM_is_a` whether an object is an instance of a given class. Once a class is obtained, a direct or indirect instance of `Class`, its fields may be read, for instance, to get its super-classes. Additionally, the `DM_is_a_subclass` predicate allows to compare classes in subclass relationship.

2.6.3 Fields

Fields may be read or written depending on their mutability characteristics. `DM_field_value` takes an object, a field descriptor (i.e., one of the fields of its class) and returns the content of the given field. If the field is indexed then `DM_indexed_field_value` takes an object, a field descriptor and an index and returns the content of the indexed field at that index. Indexes are checked to be valid for the given object. It is possible to discover the length of an indexed field with `DM_indexed_field_length` given an object and an indexed field descriptor. Indexes are zero-based.

A mutable field may be written with `DM_set_field_value` that takes an object, a field descriptor and a value. A mutable indexed field may be written with `DM_set_indexed_field_value` that takes an additional index. For that last primitive, pay attention to the order of the arguments: an indexed field is always followed by an index number; in a mutation, the new value always appears last.

As shown on figure 2.1, an indexed field is represented by a repetition prefixed by its length (*à la* Pascal strings). The encoding of the length is explicitly chosen when the class is created: it may be any of the '`nat*`" types. This type as well as the type of the value stored in a field may be obtained through the field descriptor via the `DM_field_types` function.

Accessing a (possibly indexed) field of a local object is simple: reading is allowed if the field is not secret, writing is allowed if the field is mutable. Accessing a (possibly indexed) field of a remote object is subject to the same restrictions, however, to read a volatile field involves sending a request to the owner of the object, but to read a local field is always local. Writing a local field of a (probably remote) object is always a local operation, in the other cases, writing a field of a remote object involves sending a request to perform that remote mutation. Note that reading immutable non volatile fields of a present object is always a local, therefore fast, operation.

2.6.4 Allocation

An object may be allocated with the `DM_allocate` function that takes a class and some allocation parameters to specify the length of the indexed fields. For example, a `String` is a class with a single indexed field of type `char1`, the repetition factor is a `nat3` to allow for large strings of characters. The allocation of a `String` requires one `nat3` value to be able to compute the size of the instance to allocate. Once an instance is allocated, the sizes of its indexed fields cannot be modified.

The allocation primitive is not part, *stricto sensu*, of DMeroon but must be provided by the binding language. It is often implemented with help of Hans Boehm's Garbage Collector[2]. A binding with CMM [AF94] is under study.

After an object is allocated, it must be initialized. All its fields are considered as readable or writable without the constraints required by the fields but only until the object is declared initialized. Once declared initialized, objects' fields acquire their definite characteristics, in particular, an immutable field is immutable. Two kinds of initializers exist: `DM_set_sharable` and `DM_set_copyable`. If an object is declared sharable, then it may be shared i.e., remotely referred to from another site. If an object is declared copyable then, if sent to another site, it will be copied and not shared. The remote copy will no longer be linked with the copied original object: no consistency between the two objects then exists. This is a built-in remote-cloning operation.

A third initializer exists: `DM_set_initialized` which initializes objects according to the default behavior of their class (copyable or sharable).

Once an object is initialized, it cannot be re-initialized differently: `DM_is_initialized` allows to check whether an object is initialized as well as how it was initialized i.e., sharable or copyable.

Even if classes are normal objects of class `Class`, they are not created with `DM_allocate` but with `DM_create_subclass` that takes a class and the description of a field and creates a new subclass containing that additional field. Once created the class is initialized and ready for use.

2.6.5 Site

An object is always managed by the site that owns it (by default the site where the object was allocated). The `DM_site_of` takes an object and returns the site that owns it. To migrate an object means to change the site that manages it. This may be performed by the `DM_set_site_of` function that takes an object and a site. This also impacts the associated clock since it must always be a neighbor of the object i.e., an object owned by the same site. This function belongs to level 2 and is not useful for lower levels that ignore distribution.

A site may merge its DMeroon space with the DMeroon space of another site by opening a connection between them. The `DM_connect` function takes the name of an host machine and a TCP port number and returns a reference onto the remote site object corresponding to that DMeroon server.

2.6.6 Clock

A mutable object must be associated to an instance of `Clock` if remotely shared. Such a clock is incremented any time the object is mutated. A clock may monitor multiple objects. When a site sends some information to another site, it also transmit the clocks it is aware of. A cached object may only be read if the clock that monitors it is not out of date. The clock associated to an object may be retrieved with `DM_clock_of`.

When a clock is incremented, this invalidates all the cached objects associated to that clock. It is possible to specify which clock monitors which object with `DM_set_clock_of`.

2.6.7 Communication

A site may send a reference, onto one of the objects it knows of, to another remote site. `DM_send` takes an object and a site and creates on that site a remote reference onto that object. There is no restriction on the

[2] http://reality.sgi.com/employees/boehm_mti/gc.html

object provided it is a DMEROON object.

A site may receive a reference onto an object with `DM_receive`. The object may then be ignored, accessed, modified etc. The object may convey information representing a message, a request etc.

These functions belong to level 1.

If DMEROON is used as a library then you must leave it time to manage its space, serve requests, perform garbage collection, ensure consistency, etc. This depends also on whether you use DMEROON in a sequential or multi-threaded framework. If you use DMEROON in a sequential framework then the easiest way is to call `DM_serve` with some duration from time to time. Another possibility is to call `DM_receive` with a duration of zero seconds; this also allows to empty the queue of already received objects (even if ignoring them). In a multi-threaded framework, a thread might be devoted to DMEROON.

Even if you do not plan to receive objects, you must empty the queue of received objects that may contain instances of the `Fact` class used to report asynchronous anomalies or other interesting facts such as new sites requesting a connection.

DMEROON requires the presence of an event loop to process incoming requests. The properties of this event loop depends on the precise way DMEROON is bound to your application.

- If you have a sequential binding, then any time you invoke a function of the DMEROON API then the DMEROON event loop will run while waiting for the requests needed to process your call. This means that DMEROON must respect a stack discipline: if, while processing your call, DMEROON receives a request from the network, then your call will be delayed until the network is satisfied.

- You may have a coroutine binding, at that time, your application is on top of the DMEROON event loop. Any time you invoke a function of the DMEROON API, the current thread of control (technically a continuation) is suspended, control returns to the event loop until it is appropriate to resume your call with the answer. In this mode, incoming requests are processed within their own coroutine.

  This is a coroutine mode where only one thread of control is active: when DMEROON runs, your application don't. The problem with coroutines is that passing control is explicit: a malicious thread may keep control for ever.

- You may have a multi-thread binding where DMEROON runs in its own thread as well as incoming requests. A thread, that calls the DMEROON API and requires a remote request to be satisfied, is suspended and reified into a DMEROON request object. When free, the DMEROON thread will handle it and resume after completion the waiting thread. In this binding, only one DMEROON thread may act upon the DMEROON data structures.

  This binding is still to be written.

The sequential binding is portable but less interesting than the coroutine binding which is the preferred one. Note that, with these two bindings, your application and the DMEROON event loop have different positions:

- In the sequential binding, you call `DM_serve` or `DM_receive` any time your application waits for some events from the network.

- In the coroutine binding, your application is above the DMEROON event loop: the `DM_install_receiver` API function tells DMEROON which function of your application must be run any time an object is received. With this binding, your application is triggered via a call-back function applied on every received object.

2.6.8 Equality

When objects migrate, it may be difficult to know if they are the same or not. The `DM_eq` predicate compares two objects and return true if they are the same.

2.6.9 Ubiquity

Some objects are ubiquitous i.e., local to all sites since they incarnate universal concepts. Predefined classes, DMeroon types as well as some constants (true, false) are ubiquitous. Even if DMeroon types are ubiquitous, they may have different implementation (a `reference` may have a size of 4 or 8). However it is never possible to compare two different representations of a same ubiquitous object.

2.6.10 DMeroon as a server

A DMeroon server listens for incoming connections and analyzes the first received bytes to determine the used protocol.

- If the request starts with `GET` (that is, the letters `G`, `E` and `T`) then this is supposed to start an http request. These requests represent a compact encoding of an DMeroonScript program. The DMeroonScript programming language allows to call nearly all functions of the DMeroon API. These requests may be conveniently be emitted from any HTTP client such as Netscape or IExplorer.
- If the request starts with `DMeroonScript` then the next characters represent a multi-line DMeroonScript program to be interpreted. The DMeroonScript programming language is described in chapter 5.
- If the first letters are `DMeroon` then the connection is understood as a DMeroon connection. This is the binary protocol used to marshal objects, see C.4.

It is possible to dynamically add new protocols to the list of recognized protocols: see appendix C for more details.

2.6.11 Miscellaneous

Many other functions exist in the implementation: read the source code and win the "best DMeroon undocumented feature" challenge!

Chapter 3

C binding

This chapter describes how DMeroon and C cohabits. This binding has been designed to be systematic and thread-safe.

3.1 Functions

The functions of the DMeroon API are all prefixed with `DMeroon_`, they may perform a lot of checks to ensure the validity of their arguments. Unsafer functions also exist but are hidden in the implementation.

All API functions return a status. A status of zero (known as `DME_SUCCESS`) means that everything went correctly; a non-zero status indicates the kind of problem that occurred. Other statuses exist (described in the `dmerr.h` file) and have a name prefixed by `DME_`. Error statuses are used to represent synchronous problems. Asynchronous problems (for instance, receiving unmarshable bytes from a TCP connection) cannot be reported this way to the user, they are instead reified into a `Fact` instance and received as if a remote object. It is the duty of the user to `DM_receive` these objects not to encumber memory. Another method is to install, with `DM_install_receiver`, a function to process any incoming object.

All API functions take as first argument an instance of the `Context` class i.e., the `DM_Context` structure in C. Such an instance contains a number of fields to control DMeroon behavior, see Section 3.3.

All DMeroon predefined classes without indexed fields have a related C structure whose name is the name of the class prefixed by `DM_`.

When an API function has to return a result, it must be given an address where to store this result as a side-effect: this allows to be allocation-safe. Addresses of results appear as second and successive arguments if more than one result is expected (as in `DM_field_types`). The content of these addresses is meaningful only if the API function returns `DME_SUCCESS`. It is sometimes possible to give a null pointer instead of a an address to ignore a result. This case is flagged with a `/* or NULL */` comment in the prototypes below.

As a convention for the prototypes of the DMeroon API, a variable whose name is immediately prefixed with a star indicates that it is an "out" variable that will be filled during the invocation of the function. Without a leading star, the variable is an "in" variable only. See, for instance, the `DMeroon_class_of` function below that takes an object and returns its class.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_initialize (DM_Context* context,
                    DM_Site*    *returned_site  /* or NULL */ )
```

Initializes the DMeroon library. This call must be performed before any other call to the DMeroon library otherwise undefined behavior may arise. The `Site` instance describing the current DMeroon space, i.e., the current site, is returned. It is useless but safe to call this function more than once: it always returns the current site. This is an immediate primitive. It is possible to give NULL for the returned site if not interested in it.

> **Implementation note:** *The following signature might be better since it allows to return a fresh Contexts any time it is called. This is far easier to get the first Context this way:*

```
DM_ErrorCode                                                          [Level 0]
DMeroon_initialize (DM_Context* context,           /* or NULL */
                    DM_Context* *returned_context, /* or NULL */
                    DM_Site*    *returned_site     /* or NULL */ )
```

```
DM_ErrorCode                                                          [Level 0]
DMeroon_class_of (DM_Context* context,
                  DM_Class*   *returned_class,
                  DM_Object*  object )
```

Returns the most specific class of an object that is, the class of which this object is a direct instance of. This is an immediate primitive.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_is_a (DM_Context*  context,
              DM_c_boolean *returned_boolean,
              DM_Object*   object,
              DM_Class*    class )
```

Checks whether an object is a direct or an indirect instance of a class. This is an immediate primitive. The returned boolean conforms to C practice: 0 means false, other values mean true.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_is_a_subclass (DM_Context*  context,
                       DM_c_boolean *returned_boolean,
                       DM_Class*    subclass,
                       DM_Class*    class )
```

Checks whether a class is equal to or a subclass of another one. This is an immediate primitive. The returned boolean conforms to C practice: 0 means false, other values mean true.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_field_value (DM_Context*     context,
                     t               *returned_value,
                     DM_DMeroonType* *returned_type,  /* or NULL */
                     DM_Object*      object,
                     DM_Field*       field )
```

Returns the content of a regular field of an object. This primitive may block (for instance, when reading a volatile or mutable field of a remote object). `returned_value` is the address of a cell of type *t* where *t* must be the type of the read field. This adequation may not be statically checked in C so this is an unsafe primitive. To help decoding the returned value, this primitive also returns the type of the read field that is, the DMeroon type corresponding to *t*. It is possible to give `NULL` for the returned type if not interested in.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_indexed_field_value (DM_Context*     context,
                             t               *returned_value,
                             DM_DMeroonType* *returned_type,  /* or NULL */
                             DM_Object*      object
                             DM_Field*       field,
                             DM_nat3         index )
```

Returns the content of the index'th value of an indexed field from an object. This primitive may block (for instance, when reading a volatile or mutable field of a remote object). The index is specified as a `nat3` number even if it corresponds to a `nat1`, `nat2` or `nat4` number. Indexes are therefore restricted to be inferior to 2^{32}. Indexes are checked to be valid within the object. `returned_value` is the address of a cell of type *t* where *t* must be the type of the read field. This adequation may not be statically checked in C so this is an unsafe primitive. To help decoding the returned value, this primitive also return the type of the read field that is, the DMeroon type corresponding to *t*. It is possible to give `NULL` for the returned type if not interested in.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_indexed_field_length (DM_Context*     context,
                              t               *returned_value,
                              DM_DMeroonType* *returned_type,  /* or NULL */
                              DM_Object*      object,
                              DM_Field*       field )
```

Returns the length of an indexed field. This is an immediate primitive. *t* must be the type of the repetition factor of the indexed field that is, a `nat1`, `nat2`, `nat3` or `nat4` natural number. To help decoding the returned value, this primitive also return the type of the read field that is, the DMeroon type corresponding to *t*. It is possible to give `NULL` for the returned type if not interested in.

```
DM_ErrorCode                                                                [Level 0]
DMeroon_set_field_value (DM_Context* context,
                         t           *returned_previous_value, /* or NULL */
                         DM_Object*  object,
                         DM_Field*   field,
                         t*          new_value )
```

Atomically changes the content of a regular field of an object and returns the previous content of that field. This primitive may block (for instance, when writing a mutable field of a remote object). `returned_previous_value` and `new_value` are the addresses of cells of type t where t must be the type of the written field. This adequation may not be statically checked in C so this is an unsafe primitive. When a reference is written, it is checked (as far as possible) to be the address of a DMeroon object. It is possible to give `NULL` for the returned previous value if not interested in. If t is the DMeroon `reference` type then the `new_value` must be an instance of the class specified in the `referring` field of `field`.

The `new_value` and `returned_previous_value` addresses must be different.

```
DM_ErrorCode                                                                [Level 0]
DMeroon_set_indexed_field_value (DM_Context* context,
                                 t           *returned_previous_value, /* or NULL */
                                 DM_Object*  object,
                                 DM_Field*   field,
                                 DM_nat3     index,
                                 t*          new_value )
```

Atomically changes the content of the index'th value of an indexed field of an object and returns its previous value. This primitive may block (for instance, when writing a mutable field of a remote object). The variables `returned_previous_value` and `new_value` hold the addresses of cells of type t where t must be the type of the written field. This adequation may not be statically checked in C so this is an unsafe primitive. When a reference is written, it is checked (as far as possible) to be the address of a DMeroon object. It is possible to give `NULL` for the returned previous value if not interested in. If t is the DMeroon `reference` type then the `new_value` must be an instance of the class specified in the `referring` field of `field`.

`returned_previous_value` and `new_value` must be different addresses.

```
DM_ErrorCode                                                                [Level 0]
DMeroon_field_types (DM_Context*     context,
                     DM_DMeroonType* *returned_item_type,       /* or NULL */
                     DM_DMeroonType* *returned_repetition_type,  /* or NULL */
                     DM_Field*       field )
```

Returns the type of a field as well as its repetition type if indexed. It is possible to give `NULL` for any of the two returned types when not interested by the results. If a field is not indexed, then `DM_nought_type` is returned for the repetition type (the `nought` type is characterized by a zero index).

```
DM_ErrorCode                                                      [Level 0]
DMeroon_allocate (DM_Context* context,
                  DM_Object*  *returned_object,
                  DM_Class*   class,
                  DM_nat3     number_of_sizes,
                  ... DM_nat3 size, ... )
```

Allocates an instance of a class with the specified sizes for the indexed fields. This is an immediate primitive. There must be as many sizes as there are indexed fields in the class. Sizes are given as `nat3` numbers even if they correspond to `nat1` or `nat2` or `nat4` numbers but, in that case, their value must be in the correct range compatible with `nat1` or `nat2` numbers. Indexes are therefore restricted to be inferior to 2^{32}. Due to the C limit of 32 arguments in a function call, the number of indexed fields is restricted to be less than 32; although the exact limit is fuzzy, it is probably greater than 27. The content of the object is initially filled with zeroes. If the object does not have any indexed field then the `number_of_sizes` argument is equal to zero and no other size follow it.

```
DM_ErrorCode                                                      [Level 0]
DMeroon_set_sharable (DM_Context* context,
                      DM_Object*  object )
```

Declares to DMeroon that an object is completely initialized and may now be shared over the network. This is an immediate primitive. Once an object is initialized, it cannot be re-initialized differently. When an object is sharable, a reference onto it may be sent to another site, see figure 2.2. When an object is initialized, accessing a field is constrained by the options of its field descriptor.

```
DM_ErrorCode                                                      [Level 0]
DMeroon_set_copyable (DM_Context* context,
                      DM_Object*  object )
```

Declares to DMeroon that an object is completely initialized and may now be copied over the network. A copyable object is never remotely shared, it is copied instead: there is no longer any link between the original and the copy. This is an immediate primitive. Once an object is initialized, it cannot be re-initialized differently. When an object is initialized, accessing a field is constrained by the options of its field descriptor.

```
DM_ErrorCode                                                      [Level 0]
DMeroon_set_initialized (DM_Context* context,
                         DM_Object*  object )
```

Declares to DMeroon that an object is completely initialized. The object will be sharable or copyable depending on its class. By default, classes are created with the sharable option. This is an immediate primitive. Once an object is initialized, it cannot be re-initialized differently. When an object is initialized, accessing a field is constrained by the options of its field descriptor.

```
DM_ErrorCode                                                        [Level 0]
DMeroon_is_initialized (DM_Context*  context,
                        DM_nat3      *returned_result,
                        DM_Object*   object )
```

Checks whether an object is initialized (sharable or copyable) or not. The returned result conforms to C practice: 0 means false, other values mean true. However the result is more informative as it tells whether the object is sharable or copyable. Results are then, respectively, `DM_OBJECT_IS_SHARABLE` or `DM_OBJECT_IS_COPYABLE`. This is an immediate primitive.

```
DM_ErrorCode                                                        [Level 0]
DMeroon_create_subclass (DM_Context*      context,
                         DM_Class*        *returned_class,
                         DM_Class*        superclass,
                         char*            class_name,
                         DM_class_options class_options,
                         char*            field_name,
                         DM_field_options field_options,
                         DM_DMeroonType*  repetition_type,
                         DM_DMeroonType*  item_type,
                         DM_Class*        referring_class /* or NULL */
                        )
```

This primitive creates a new class with a name and some class options, that adds one additional field to the super class. The field is specified with a name and some field options. One must also specify its type and whether it is indexed or not. If the type is the `reference` type then the class, the values should be instance of, must be given. This primitive may block.

Possible class options belong to the `DM_class_options` type. They are summarized in table 3.1. Some of these options apply to the class (this is the case of `DM_CLASS_IS_VOLATILE` or `DM_CLASS_IS_MUTABLE`), other options apply on instances (this is the case of `DM_CLASS_IS_IMMOTILE` and `DM_CLASS_IS_COPYABLE`). Still other options are only used within DMeroon implementation.

name	**meaning**
`DM_CLASS_DEFAULT_OPTIONS`	Instances are motile, immutable, sharable, non-volatile.
`DM_CLASS_IS_IMMOTILE`	The class is immotile. Its instances cannot be migrated.
`DM_CLASS_IS_COPYABLE`	The class is copyable. Its instances cannot be shared unless they are initialized with `DM_set_sharable`.
`DM_CLASS_IS_NOT_SUBCLASSABLE`	The class cannot be subclassed.
`DM_CLASS_IS_VOLATILE`	The class is volatile: at least one fields is volatile.
`DM_CLASS_IS_MUTABLE`	The class is mutable: at least one field is mutable.
`DM_CLASS_IS_NOT_INSTANTIABLE`	The class cannot be instantiated.

Table 3.1: Class options

Possible field options belong to the `DM_field_options` type. They are summarized in table 3.2.

> **Implementation note:** *Should offer a new type of field: guarded field. A guarded field somewhat extends secret fields. To be read, a correct password must appear in the current context. The comparison is of course done only on the owning site. Such fields are like portals. Should*

name	meaning
DM_FIELD_DEFAULT_OPTIONS	The field is immutable, non volatile, non local and non secret.
DM_FIELD_IS_MUTABLE	The field is mutable, the user may modify it.
DM_FIELD_IS_VOLATILE	The field is volatile, its value may evolve spontaneously.
DM_FIELD_IS_LOCAL	The field is local, its content is never marshaled.
DM_FIELD_IS_SECRET	The field is secret, its content cannot be read.

Table 3.2: Field options

think to how to have more than one password in the context. The context must also be passed from site to site when emitting requests.

I also consider adding the DM_FIELD_IS_RESIZABLE *option. This complicates the coherency protocol but makes users' life far easier. Of course, a resizable field should be mutable.*

The previous primitive may only define additional fields one by one (this is needed to keep the prototype of the C function static) but this increases considerably the depth of the tree of classes. That's why there is another primitive to create classes.

[*Level 0*]

```
DM_ErrorCode
DMeroon_smash_class (DM_Context*      context,
                     DM_Class*        *returned_class,
                     DM_Class*        class,
                     char*            class_name,
                     DM_class_options class_options,
                     DM_nat3          level )
```

This primitive returns a new class with a name and some class options. This new class has exactly the same fields the given class has but only a shortened sequence of super-classes i.e., level superclasses less. This primitive may block. This primitive is used with the previous one to create a direct subclass with multiple fields.

For instance, to create a direct subclass of A with two additional fields, say x and y, you first create B as $A+x$, then C as $B+y$. C has the required fields but has, as super-classes A, B whereas only A is wanted. To remove the B super-class, we smash one level of super-classes from C and obtain D as $A+x+y$. The x and y fields are assumed to have been introduced by D.

[*Level 0*]

```
DM_ErrorCode
DMeroon_site_of (DM_Context* context,
                 DM_Site*    *returned_site,
                 DM_Object*  object )
```

Returns the site that owns the object. This primitive may block. The current site may be obtained with `DM_initialize`. Ubiquitous objects always have the current site as site.

```
DM_ErrorCode                                    [Level 2 — NOT YET IMPLEMENTED]
DMeroon_set_site_of (DM_Context* context,
                     DM_Site*    *returned_previous_site,
                     DM_Object*  object,
                     DM_Site*    new_site )
```

Changes the site that owns an object and returns the previous owner; this is also called "migration". The object must be motile otherwise its owner cannot be altered and the object cannot move out of the site where it had been created (this does not prevent the object to be copied or shared that is, remotely referenced). This primitive may block. This primitive has an impact on the monitoring clock if any.

```
DM_ErrorCode                                                          [Level 0]
DMeroon_clock_of (DM_Context* context,
                  DM_Clock*   *returned_clock,
                  DM_Object*  object )
```

Returns the instance of `Clock` that monitors a mutable object. This is an immediate primitive. If the object is immutable or not initialized or not monitored then `DM_NULL` is returned instead. When creating a remote reference onto a mutable object which is not monitored, the general clock of the current site is assumed to monitor it by default.

```
DM_ErrorCode                                  [Level 2 — NOT FULLY IMPLEMENTED]
DMeroon_set_clock_of (DM_Context* context,
                      DM_Clock*   *returned_previous_clock,
                      DM_Object*  object,
                      DM_Clock*   clock )
```

Sets or changes the clock monitoring a mutable object and returns the previous clock if there was one or `DM_NULL`. The object may be not initialized. The new clock must be a neighbor of the object that is, owned by the same site. This primitive may block.

```
DM_ErrorCode                                                          [Level 1]
DMeroon_connect (DM_Context* context,
                 DM_Site*    *returned_site,
                 char*       hostname,
                 DM_nat2     port_number,
                 char*       passwd )
```

Connects the current site to the given remote site. This primitive may block. It returns a reference onto the remote site object. The hostname may be the human-readable name of a machine or an IP number in dot notation. The port number must be a regular natural number (do not specify it in network order (don't care if you do not know what is network order, just give a number)).

> **Implementation note:** *Should probably think again to this password stuff. Would be better to use SSL to protect communications.*

```
DM_ErrorCode                                                    [Level 1]
DMeroon_send (DM_Context* context,
              DM_Object*  object,
              DM_Site*    site )
```

Sends the reference onto an object to a site. This primitive may block.

> **Implementation note:** *The reference is immediately marshaled in an* `OutBuffer` *that will be flushed when the Connection will allow it. The reference will be unmarshaled in a queue of arrived objects. Therefore when returning from* `DMeroon_send` *there is no guarantee that the reference is obtained by the intended site.*

```
DM_ErrorCode                                                    [Level 1]
DMeroon_receive (DM_Context* context,
                 DM_Object*  *returned_object,
                 DM_nat3     duration )
```

Receives an object. This primitive blocks at most *duration* seconds. If no object is received during this time then it returns the `DME_TIMEOUT` error code. DMeroon signals asynchronous errors by reifying them into instances of `Fact`. These Facts may be normally received. This primitive is exclusive of `DMeroon_install_receiver`.

```
DM_ErrorCode                                                    [Level 1]
DMeroon_install_receiver (DM_Context* context,
                          void(*      handler)(DM_Context*, DM_Object*) )
```

This primitive is exclusive of `DMeroon_receive`. The `DMeroon_install_receiver` function tells DMeroon to invoke a given function on every incoming object. The context received by the handler function is the one given to `DMeroon_install_receiver`. By default, the receiver function accumulates incoming objects in a field of the current site from which they may be extracted using `DMeroon_receive`.

```
DM_ErrorCode                                                    [Level 0]
DMeroon_serve (DM_Context*  context,
               DM_nat3      duration )
```

This primitive yields control to DMeroon for at least *duration* seconds. Zero seconds is a possible duration in which case, pending requests are handled if already present. This allows DMeroon to handle incoming requests and manage its local space. If a receiver function is installed, then any incoming object will be automatically submitted to that receiver function, otherwise, you must `DMeroon_receive` these objects explicitly after returning from `DMeroon_serve`.

> **Implementation note:** *As a kind of answer to the need for interfaces, I consider adding virtual fields.*

`VirtualField` [*Class*]

`name` a repetition of characters

...

A virtual field allows to gather instances of `Field` *(or* `VirtualField`*) in order to provide an unified access to many unrelated instances of unrelated classes. For instance, I imagine the* `name` *virtual field that allows access the* `name` *field of* `Class`*,* `Field`*,* `Site`*,* `Type`*, etc. To get the name of instances of any of these classes, you may just use the regular* `DMeroon_field_value` *function with the virtual field instead of the appropriate instance of* `Field`*. The appropriate field will be automatically chosen in the set of fields contained in the virtual field. Virtual fields are similar to generic functions la CLOS (note: I will probably introduce the discrimination mechanism directly so it can be used in other places, for instance, to implement generic functions.)*

`VirtualField` *instances are immutable. They have a name for introspection. They may be extended to incorporate new fields with the* `DMeroon_extend_virtual_field` *function.*

3.2 Global variables

In order to instrospect DMeroon objects, you need at hand the predefined classes, their associated fields as well as the DMeroon portable types. They are all accessible via specialized macros (available from the `dmeroon.h` header file). The case is significant: Classes have a capitalized name, Fields are written in lower-case, and components are separated with underscores.

The files that are mentioned below are relative to the *TOP* directory where the source files are. More on these files in appendix A.

`DM_ReferenceToClass(`*name*`)` [*Level 0*]

This macro takes the name of a class and returns a DMeroon reference onto a predefined `Class`, the returned value has `DM_Class*` type for C. Most predefined class names appear in table 2.3 however they all appear in the *TOP*`/DMeroon/c/dmstruct.dm` file.

`DM_ReferenceToFieldOfClass(`*field-name*`, `*class-name*`)` [*Level 0*]

This macro takes the name of a field and the name of a class and returns a DMeroon reference onto a predefined instance of `Field`; the returned value has `DM_Field*` type for C. Some fields were mentioned in section 2.5.1 otherwise they all appear in the *TOP*`/DMeroon/c/dmstruct.dm` file.

`DM_ReferenceToDMeroonType(`*name*`)` [*Level 0*]

This macro takes the name of a DMeroon portable type and returns a DMeroon reference onto a predefined instance of `DMeroonType`; the returned value has `DM_DMeroonType*` type for C. DMeroon types are listed in table 2.1.

3.3 Contexts

<< Section under construction >>

Contexts control many aspects of DMeroon behavior. Where alternate behaviors are possible, they may be specified by the user by toggling some options of the context. When calling the DMeroon API, a context should be given if the user has peculiar needs. If no context is given, a default one is cloned after the default context held in a field of the current site. Binding languages may customize this default context.

Contexts are threaded throughout DMeroon innards, there is (nearly) always a current context everywhere. The context contains options that may be trigger to toggle various behaviors. Whenever an exception occurs, reified by an error code, the context provides a handler that will be invoked to manage this exception. The handler may be customized to take benefit of the support language.

A context is an instance of class `Context` which contains the following fields:

`Context` [*Class*]

`handler`	*a C function to handle exceptions*
`options`	*a sequence of bits encoded as a* `context-options`
`errcode`	*the last* DMeroon *error code*
`file`	*the name of the file of the last error (a C string)*
`line`	*the line number within that file*
`errnumber`	*the last* `errno` *code (not portable)*
`previous`	*previous context (for debug purpose)*
`continuation`	*where to go in case of error*
`errstream`	*where to report errors*

These fields are local and volatile. They are mainly used by the implementation to report errors. These fields are not reset when a DMeroon API function is called. The `errstream` field contains the port where warnings or errors are reported, the exact meaning of the port depends on the binding language. For C and Un*x, stdout and stderr are legal values; Scheme ports are also legal for Bigloo.

The first field designates the function to handle the exception. This function must have the following prototype:

```
void (*handler)(DM_Context*   context,
                DM_ErrorCode  errcode,
                char*         __FILE__,
                unsigned long __LINE__ )
```

The third and fourth arguments show the birth place of the exception. The second argument is the error code (see section later). The first argument is the current context where additional information may be found (such as the value of `errno`) set by various system calls. By default, this function is the `DM_abort` function that notifies the problem then escape to a safer place as indicated in the `continuation` field (a `jmpbuf` for C, a real continuation for Scheme).

The following options are legal:

`DM_CONTEXT_CHECK_DMEROON`	Forces DMeroon to check every object received as argument to be a DMeroon object.
`DM_CONTEXT_CHECK_CLASS`	Forces DMeroon to check every object received as argument to be an instance of the appropriate class.
`DM_DONT_INITIALIZE`	When DMeroon allocates an object, this object is by default initialized. When set, DMeroon leaves them uninitialized so it is up to the user to make them sharable or copyable.

There are other hidden options that make it is possible, in case of anomalies, to force DMeroon to `longjmp` to a given position or to call an internal debugger.

3.4 Error codes

<< Section under construction >>

Hundreds of error codes exist. They are detailed in the *TOP*`/DMeroon/c/dmerr.h` file.

3.5 Static generation of classes

If you consider defining classes and writing C code around these classes (for instance, to contribute new code to DMeroon), you may want to use static classes and their associated C structs to avoid creating these classes dynamically. This is contrary to the spirit of DMeroon but you may require these classes to be immutable, volatile, local and immotile so you can safely hack them on the current site. There exists a compiler that takes the definition of these classes and produces the appropriate C files.

The current compiler is written in Scheme, its name is `dm2ch` which is the name of a shell script (from *TOP*`/DMeroon/Commands`) that must be run in the *TARGET* directory (the directory were all files are compiled, see again appendix A). This compiler is used by DMeroon itself for its own bootstrap.

`dm2ch` *file other-files...* [*Command*]

The first file contains the classes to be compiled. These classes may inherit from other classes which are not to be compiled but which must be known at compilation-time (the DMeroon classes, for instance, which are defined in the *TOP*`/DMeroon/c/dmstruct.dm` file, see appendix A). These classes should appear in the other files mentioned in the command line.

The compiler produces two files: a *file*`.c` and a *file*`.h` defining the DMeroon representation of the desired classes. The *file*`.h` file contains the equivalent C structs and can be included in your C code. The C file must be compiled and linked to your code: it only contains static data and no code at all. You may refer to these classes and their fields with the macros of section 3.2.

Classes are defined (as in MeroonV3) that is, as Sexpressions:

(`define-class` *classname superclassname* [*Scheme macro*]
((*fieldname field-options*)
...)
class-options)

Classes are named shortly i.e., as `Object`, `QueueLink`, etc. DMeroon types are also named shortly i.e., as `nat3`, `char1`, etc. All fields must have a name and be qualified by some options as described in table 3.3.

The class itself may be qualified by some options as described in table 3.4.

The compiler will warn you if it cannot generate a C struct that emulates a DMeroon class. This may be the case if you have a class with fields following an indexed field.

3.6 Tutorial examples

<< Section under construction >>

option	**meaning**
`:type` *DMeroonType*	This option is mandatory. The type of the field must be one of the DMeroon types of figure 2.1.
`:indexing-type` *DMeroonType*	By default, a field is not indexed. Mentioning `:indexing-type` requires to specify a type among `nat1`, `nat2`, `nat3` and `nat4`.
`:refer` *class-name*	This option is only meaningful for the `reference` type. It specifies the class of referred objects. Default is `:refer Object`.
`:mutable`	The field is mutable. No option means that the field is immutable.
`:local`	The field is local.
`:volatile`	The field is volatile.
`:secret`	The field cannot be read.

Table 3.3: Field options

option	**meaning**
`:immotile`	By default classes are motile. This option prevents instances of that class to migrate.
`:mutable`	At least one field is mutable. This option is automatically set.
`:volatile`	at least one field is volatile. This option is automatically set.
`:copyable`	instances of this class are, by default, copied rather than shared.
`:uninstantiable`	this class cannot be instantiated.
`:unsubclassable`	this class cannot be subclassed.

Table 3.4: Class options

3.6.1 Dynamic creation

Here is how you may define the class `Point`. We extend `Object` with the two fields `x` and `y`, then we smash the resulting class off a level. To lessen the size of the example and although this is highly incorrect, error codes are not checked!

```
DM_Class *point_class, *px, *py;
  (void)DMeroon_create_subclass(ctx, &px, DM_ReferenceToClass(Object),
                                "no name", DM_CLASS_DEFAULT_OPTIONS,
                                "x", DM_FIELD_IS_MUTABLE,
                                  DM_ReferenceToDMeroonType(nought),
                                  DM_ReferenceToDMeroonType(int3),
                                  DM_NULL );
  (void)DMeroon_create_subclass(ctx, &py, px,
                                "no name", DM_CLASS_DEFAULT_OPTIONS,
                                "y", DM_FIELD_IS_MUTABLE,
                                  DM_ReferenceToDMeroonType(nought),
                                  DM_ReferenceToDMeroonType(int3),
                                  DM_NULL );
  (void)DMeroon_smash_class(ctx, &point_class, py, "Point",
                                 DM_CLASS_DEFAULT_OPTIONS, 1);
```

Now the `Point` class is the value of the `point_class` variable. Its first field may be obtained as follows (and similarly for the second field):

```
DM_Field *point_x_field;
  (void)DMeroon_indexed_field_value(ctx, &point_x_field, DM_NULL,
                                    point_class,
                                    DM_ReferenceToFieldOfClass(field, Class),
                                    0 );
```

Once the class is created, an instance may be created, filled and initialized like that:

```
DM_Object* pt;
DM_int3    ix = -12345;
(void) DMeroon_allocate(ctx, &pt, point_class, 0);
(void) DMeroon_set_field_value(ctx, NULL, pt, point_x_field, &ix);
(void) DMeroon_set_initialized(ctx, pt);
```

3.6.2 Static definition

This is far more simple and may even be made inline with DMeroon bound with Scheme. First, create a file containing the definition of the `Point` class.

```
;;; This is file Point.dm
(define-class Point Object
  ((x :type int3 :mutable)
   (y :type int3 :mutable) ) )
```

Second, compile this file with the `dm2ch` compiler. The command is (it is not necessary to mention *TOP*`/DMeroon/c/dmstruct.dm` since `Point` only inherits from the (empty) `Object` class):

```
dm2ch Point.dm
```

This compilation produces the `Point.h` and `Point.c` files. You then have to compile them with your favorite C compiler and to link the resulting `Point.o` file to your application. You may take advantage of this class in your other files and refer to the class or its fields writing (the C type of these constructions is given in a prefixed comment):

```
#include "Point.h"   /* Don't forget this inclusion! */
/* DM_Class* */      DM_ReferenceToClass(Point)
```

```
/* DM_Field* */      DM_ReferenceToFieldOfClass(x, Point)
/* DM_Field* */      DM_ReferenceToFieldOfClass(y, Point)
```

3.6.3 Recursive class

We now want to create a subclass of `Point`, say `LinkedPoint`, with an additional field which is a reference onto an instance of `LinkedPoint`.

<< Section under construction >>

Chapter 4

Scheme binding

This chapter describes how DMeroon and Scheme cohabit. This binding has been designed to be Scheme-friendly: errors are reported via exceptions when the particular Scheme dialect allows it; results are returned by function calls. This is a far more natural and user-friendly interface compared to the C binding.

> **Implementation note:** *This binding worked with Bigloo 1.8. It looks like working with a pre-alpha release of Bigloo 1.9d. It does not work with the current distribution of Bigloo.*

When accessing objects, DMeroon offers an interface not too dissimilar to Meroon [Que93].

All API names to be used from Scheme are prefixed with `DM-`. They are implemented by global variables or C functions (prefixed with `DMscm_` and defined in the *TOP*`/DMeroon/c/dmscm.c` file). They use a set of macros to convert C values to Scheme values back and forth. This set of macros respect the usual conventions of Scheme implementations for value representation. Technically, this layer uses the C binding layer. It has been used to bind DMeroon with Bigloo, OScheme and PicoLisp. Scheme values are automatically coerced into DMeroon values back and forth according to table 4.1. Range checks are also performed.

Scheme type	DMeroon **type**
integer	`nat1 nat2 nat3 nat4 int1 int2 int3 int4 netnat2 netnat3`
float	`float1 float2 float3`
DMeroon object	`reference`
value	`data`
character	`char1`

Table 4.1: Scheme–DMeroon coercions

Here are the conventions that are used below. A capitalized name such as *Object* or *Class* means a value that is an instance of the given class. The word *value* means any Scheme value whereas *boolean* means only `#t` or `#f`. Natural numbers i.e., positive integers are represented by words such as *length* or *index*. These words are sometimes prefixed by *new-* or *old-* to disambiguate some of their occurrences. Scheme strings are enclosed within double quotes with a description of their content.

4.1 Functions

`(DM-initialize!) :` *Site* [*Level 0*]

This primitive returns the current site (this object is also the value of the `DM-site` variable). The DMeroon library is already initialized by the implementation. This primitive is immediate.

`(DM-is-object?` *value*`)` : *boolean* [*Level 0*]

Checks whether a Scheme value is a DMeroon object or not. This is an immediate primitive.

`(DM-class-of` *Object*`)` : *Class* [*Level 0*]

Returns the most specific class of an object that is, the class of which this object is a direct instance of. This is an immediate primitive.

`(DM-is-a?` *Object Class*`)` : *boolean* [*Level 0*]

Checks whether an object is an instance of a class. This is an immediate primitive.

`(DM-is-a-subclass?` *Class Class*`)` : *boolean* [*Level 0*]

Checks whether the first argument is equal to or a subclass of the second argument. This is an immediate primitive.

`(DM-field-value` *Object Field*`)` : *value* [*Level 0*]

Returns the content of a regular field of an object. This primitive may block (for instance, when reading a volatile or mutable field of a remote object). The value is automatically coerced into a Scheme value according to table 4.1.

`(DM-indexed-field-value` *Object Field index*`)` : *value* [*Level 0*]

Returns the content of the index'th value of an indexed field of an object. This primitive may block (for instance, when reading a volatile or mutable field of a remote object). The value is automatically coerced into a Scheme value if possible.

`(DM-indexed-field-length` *Object Field*`)` : *length* [*Level 0*]

Returns the length of an indexed field. This is an immediate primitive.

`(DM-set-field-value!` *Object Field new-value*`)` : *old-value* [*Level 0*]

Atomically changes the content of a regular field of an object and returns the previous content of that field. Values are coerced back and forth between DMEROON and Scheme according to table 4.1. This primitive may block (for instance, when writing a mutable field of a remote object). When a reference is written, it is checked (as far as it is possible) to be the address of a DMEROON object of an appropriate class.

`(DM-set-indexed-field-value!` *Object Field index new-value*`)` : *old-value* [*Level 0*]

Atomically changes the content of the index'th value of an indexed field of an object and returns its previous value. Values are coerced back and forth between DMEROON and Scheme according to table 4.1. This primitive may block (for instance, when writing a mutable field of a remote object). When a reference is written, it is checked (as far as it is possible) to be the address of a DMEROON object of an appropriate class.

Pay attention to the order of the arguments. The index always follow the field (since it is an indexed field) and the value is always given last.

`(DM-field-item-type` *Field*`)` : *DMeroonType* [*Level 0*]

Returns the type of a value stored in a Field (whether indexed or not). This is an immediate primitive.

`(DM-field-repetition-type` *Field*`)` : *DMeroonType* [*Level 0*]

Returns the type of the repetition factor of a Field. If the Field is not indexed then `DM-nought-type` is returned instead. This is an immediate primitive.

`(DM-allocate` *Class sizes…* `)` : *Object* [*Level 0*]

Allocates an instance of a class with the specified sizes for the indexed fields. This is an immediate primitive. There must be as many sizes as there are indexed fields in the class. The content of the object is initialized with zeroes.

`(DM-set-sharable!` *Object*`)` : *the Object itself* [*Level 0*]

Declares to DMeroon that an object is completely initialized and may be shared over the network. This is an immediate primitive.

`(DM-set-copyable!` *Object*`)` : *the Object itself* [*Level 0*]

Declares to DMeroon that an object is completely initialized and may be copied over the network. This is an immediate primitive.

`(DM-set-initialized!` *Object*`)` : *the Object itself* [*Level 0*]

Declares to DMeroon that an object is completely initialized. It will be shared or copied depending on its class. This is an immediate primitive.

`(DM-is-initialized?` *Object*`)` : *boolean* [*Level 0*]

Returns true or false whether the object is initialized (sharable or copyable) or not. In fact, the result when true is more informative, 1 is returned when the object is sharable while 2 is returned if the object is copyable. Once initialized, an object cannot be re-initialized differently.

`(DM-create-subclass` *Class* "*class name*" *class-options* [*Level 0*]
"*field name*" *field-options*
DMeroonType DMeroonType Class `)` : *Class*

This primitive creates a new class with a name and some class options. The newly defined class adds one additional field to its super class. The field is specified with a name and some field options. One should also specify its type and whether it is indexed or not. If the type of the field is the DMeroon `reference` type then the final argument, a class, specifies the class the values must be instance of. This primitive may block.

Class-options are represented by integers with the C encoding. They are summarized on table 4.2. Add these values to specify more than one.

code	**C name**	**meaning**
0	`DM_CLASS_DEFAULT_OPTIONS`	Instances are motile, immutable, sharable, non-volatile.
1	`DM_CLASS_IS_MUTABLE`	The class is mutable: all fields are mutable.
2	`DM_CLASS_IS_IMMOTILE`	The class is immotile. Its instances cannot be migrated.
8	`DM_CLASS_IS_VOLATILE`	The class is volatile: all fields are volatile.
64	`DM_CLASS_IS_NOT_SUBCLASSABLE`	The class cannot be subclassed.
128	`DM_CLASS_IS_COPYABLE`	The class is copyable. Its instances cannot be shared unless initialized with `DM_set_sharable`.
256	`DM_CLASS_IS_NOT_INSTANTIABLE`	The class cannot be instantiated.

Table 4.2: Class options

Field-options are similarly represented as shown on table 4.3. Add these values to specify more than one.

code	C name	meaning
0	`DM_FIELD_DEFAULT_OPTIONS`	The field is immutable, non volatile and non local.
1	`DM_FIELD_IS_MUTABLE`	The field is mutable.
2	`DM_FIELD_IS_VOLATILE`	The field is volatile.
4	`DM_FIELD_IS_LOCAL`	The field is local.
16	`DM_FIELD_IS_SECRET`	The field is secret.

Table 4.3: Field options

`(DM-smash-class` *Class* `"`*class name*`"` *class-options level*`) :` *Class* [*Level 0*]

This primitive returns a new class with a name and some class options. This new class has fields similar to the given class but it has level superclasses less. This primitive may block. This primitive is used with the previous one to allow to create a direct subclass with multiple fields. However, classes may be more easily created with the `DM-define-class` macro (see Section 4.3.2).

`(DM-site-of` *Object*`) :` *Site* [*Level 0*]

Returns the site that owns the object. This is an immediate primitive.

`(DM-set-site-of!` *Object new-Site*`) :` *old-Site* [*Level 2 — NOT YET IMPLEMENTED*]

Changes the site that owns an object and returns the previous owner; this is also called "migration". The object must be motile otherwise its owner cannot be altered and the object cannot move from the site where it had been created (this does not prevent the object to be copied or shared that is, remotely referenced). This primitive may block. This primitive impacts the monitoring clock if any.

`(DM-clock-of` *Object*`) : #f` *or Clock* [*Level 0*]

Returns the clock monitoring a mutable object. This is an immediate primitive. If the object is immutable or not monitored by a clock then `#f` is returned instead.

`(DM-set-clock-of!` *Object new-Clock*`) :` *old-Clock* [*Level 2 — NOT YET IMPLEMENTED*]

Changes the clock associated to a mutable object and returns the previous clock. The new clock must be a neighbor of the object that is, owned by the same site. This primitive may block.

`(DM-connect "`*host name or IP number*`"` *port-number* `"`*password*`") :` *Site* [*Level 1*]

Connects the current DMEROON space with a remote one. The remote DMEROON space is specified by an hostname and a port number. The hostname may also be an IP number written in dot notation.

`(DM-send` *Object Site*`) :` *the Object itself* [*Level 1*]

Sends a reference onto an object to a site. This primitive may block.

`(DM-receive` *duration*`) : #f` *or an Object* [*Level 1*]

Receives an object. This primitive blocks at most *duration* seconds. If no object is received then it returns false. This function may only be used while the `DM-handler-object` variable is false.

`(DM-handler-object` *Object*`) : #f` [*Level 1*]

If this variable is false i.e., `#f` then incoming objects are accumulated then received using `DM-receive`. If this variable is not false, then its value must be a unary function that will be applied on every incoming object. In this case, one must not call `DM-receive`.

`(DM-serve` *duration*`) :` *bolean* [*Level 0*]

This primitive yields control to DMEROON for at least *duration* seconds. Zero seconds is a possible duration in which case, pending requests are handled if already present. This allows DMEROON to handle incoming requests and manage its local space.

4.2 Variables

The current site is the value of the `DM-site` variable.

A number of global variables allow to access predefined classes. Their name start with `DM-` and end with `-class`. These are, for instance, `DM-Object-class`, `DM-Class-class`, `DM-Field-class`, `DM-Type-class`, `DM-String-class`, etc.

To allow an easy access to instances of predefined classes, fields of predefined classes are also predefined. Their name begin with `DM-`, the name of the introducing class, a dash, the name of the field and is followed by `-field`. These are, for instance, `DM-Class-name-field`, `DM-Class-super-field`, `DM-Class-field-field`, etc.

DMeroon types are also defined under a name ending with `-type`. These are, for instance, `DM-nou-ght-type`, `DM-nat1-type`, `DM-nat2-type`, `DM-nat3-type`, etc.

4.3 Libraries

The Scheme binding defines additional useful functions. These files may be dynamically loaded or compiled and linked.

4.3.1 Tools

Some useful functions are defined in the *TOP*/`DMeroon/dmlib.scm` file. Among them is an iterator on objects: `DM_iterate`, a coercer into Scheme strings, a function to display details of DMeroon objects, functions to convert Scheme values into DMeroon objects back and forth, etc.

(`DM-iterate` *Object by-item by-length by-indexed-item*) : *unspecified* [*Scheme function*]
(*by-item Object Field DMeroonType*) : *unspecified*
(*by-length Object Field DMeroonType DMeroonType*) : *unspecified*
(*by-indexed-item Object Field DMeroonType index*) : *unspecified*

Iterate sequentially on all the non-secret values contained in an object (but the secret ones) and apply appropriate functions on these values. The *by-item* function is applied on every regular non-secret field, the *by-length* function is applied on every non-secret indexed field, the *by-indexed-item* function is applied on every value of every non-secret indexed field.

(`DM->string` *Object [Field]*) : *String* [*Scheme function*]

This function takes an instance of the `String` class and coerces it into a Scheme string. Used with two arguments: an object and an additional instance of the `Field` class corresponding to a repetition of `char1`, it coerces the field into a Scheme string. For example, obtaining the name of the current site may be written as:

```
(DM->string DM-site DM-Site-name-field)
```

Implementation note: *A common error is to confuse Scheme and* DMeroon *values, especially strings and instances of* `String`*. Use the* `DM->string` *or* `DM<-sexpr` *functions!*

(`DM-describe` *Object [output-port]*) : *unspecified* [*Scheme function*]

This function prints a detailed description of an object onto a Scheme output port (by default onto the current output port). Only the first level of the object is printed, use the next function to recursively display an object. This function is implemented with `DM-iterate`.

(`DM-describe-all` *Object [output-port]*) : *unspecified* [*Scheme function*]

This function prints a detailed description of an object onto a Scheme output port (by default onto the current output port), it also recursively prints the detailed description of referenced objects. Cycles of objects are handled. This function is implemented with `DM-iterate`.

`(DM<-sexpr` *value [handler]*`)` [*Scheme function*]

This function converts a Scheme value into a DMeroon object. The empty list, booleans, symbols, characters, vectors, pairs, strings, integers and floats are converted. Other types of values (procedures for instance) are given to the *handler*, their translation is left to the definition of the handler. The default handler raises an error. Sharing is not respected by this simple-minded converter but DMeroon objects are left as they are.

`(DM->sexpr` *value [handler]*`)` [*Scheme function*]

This function converts a DMeroon object into a Scheme value. The empty list, booleans, symbols, characters, vectors, pairs, numbers and strings are converted. If a DMeroon object is not converted then it is given to the *handler* which, by default, raises an error. Sharing is not respected by this simple-minded converter.

`(DM-publish!` *Object* `"`*name*`" ...) :` *unspecified* [*Scheme function*]

This function publishes a DMeroon object i.e., incorporates it in an instance of the *Dictionary* class in the offspring of the site object (more precisely via the `information` field of the current site). The pathname leading to this object is indicated in the following arguments of the `DM-publish!` function. By default, every value published by the user is in the `user` dictionary.

Published objects can be inspected with an URL formed after their pathname.

`(DM-retrieve "`*name*`" ...) :` *Object or* `#t` [*Scheme function*]

This function retrieves the DMeroon object that is published under the given pathname. If that object does not exist then `#f` is returned instead.

To store my first name, I can say:

```
(DM-publish! (DM<-string "Queinnec") "name" "first")
```

It can now be retrieved with:

```
(DM-retrieve "name" "first")
```

4.3.2 DMeroonet

The *TOP*`/DMeroon/dmeroonet.scm` file defines a light object system, above DMeroon, in the spirit of MeroonV3.

The DMeroonet object system is very simple to use: only three macros are defined: one to define classes (this macro is almost similar to the `define-class` macro described in Section 3.5). DMeroonet provides distributed generic functions i.e., generic functions whose methods may be located on different sites [Que97a].

(DM-define-class *class-name super-class* [*Scheme macro*]
(*field-name field-options*)
[class-options])

This macro defines a class with a name (a symbol), a super-class (an evaluated expression that must yield a class), some fields and, possibly, some class options. Class options are described in table 3.4. A field starts with a name (a symbol) followed by field options as described in table 3.3.

For example, here is the definition of the `Point` class (note the difference with the definition of Section 3.6.2):

```
;;; This is file Point.dm
(DM-define-class Point DM-Object-class
  ((x :type int3 :mutable)
   (y :type int3 :mutable) ) )
```

(DM-generic (*name variables-specification*) [*Scheme macro*]
[default body])

This macro defines an anonymous generic function. The *variables-specification* specifies the variables of the function as well as which variables are discriminating. A discriminating variable appears between parentheses. The values of the discriminating variables allow to choose the appropriate method. By default, the default behavior of a generic function is specified by the *default body*. In particular, if applied on Scheme values rather than DMeroon values, the default body is applied. The generic function itself is a DMeroon object value of the *name*; the default body may be recursive and use this anonymous generic function with the *name* variable.

(DM-method (*name variables-specification*) [*Scheme macro*]
[body])

This macro defines a method for a generic function. The generic function may be referred to with the *name* variable. The *variables-specification* specifies the variables of the function as well as which variables are discriminating. Of course, this must be congruent with the generic function(s) to which this method will be added. A discriminating variable appears between parentheses accompanied by an expression whose value is the class on which the method will be triggered.

(DM-add-methods *generic-function method...*) [*Scheme function*]

This function allows to add multiple methods to a generic function. It returns a new generic function similar to the previous one except that it understands new behaviors.

`(DM-invoke` *generic-function value...*`)` [*Scheme function*]

This function invokes a generic function on some arguments. The arguments that correspond to discriminating variables allow to choose the appropriate method which is then applied on these arguments.

`(call-next-method)` [*Method macro*]

This macro can only appear in a `method` definition. It allows to call the super method on the same arguments.

`(call-former-method` *value...*`)` [*Method macro*]

This macro can only appear in a `method` definition. It allows to refer to the method that would have been triggered in the generic function that was enriched by `DM-add-methods`.

4.4 Scheme peculiarities

The above documentation describes the general binding of DMeroon with a generic Scheme implementation. Often it is possible to bind it better. This section describes these additions.

4.4.1 Bigloo

The binding of DMeroon with Bigloo 1.8 brings some additional features.

1. the `display` function is extended to recognize DMeroon objects. They are printed as

   `#<DMeroon:`*reference*`>`

   where *reference* is a DMeroon reference (and not a Bigloo address).

2. if Bigloo waits when reading a file or the input terminal then the interpreter will take advantage of that situation to serve incoming DMeroon requests. DMeroon uses Bigloo's continuations to create or, suspend or, resume coroutines. With `DM-handle-object`, any incoming object is processed in its own coroutine.

3. a new protocol is recognized: the `SchemeScript` protocol which allows to receive Sexpressions to evaluate via the DMeroon server. These Sexpressions are evaluated in the global environment of the interpreter. The displayed result is sent back to the client and the connection is closed. A `SchemeScript` message ends with a sequence of four characters which are the NUL, NUL, CR and LF characters in that order.

To use that protocol, you may use the `schemescript` command defined in the *TOP*`/DMeroon/Commands/schemescript.prl` file.

`schemescript.prl` *[host[:port] [file]]* [*Command*]

This command sends the content of the given *file* (or the standard input if absent) to the DMeroon server specified by *host:port*. It prints the returned result but does not report errors (they are reported via the standard error output of Bigloo).

Chapter 5

DMeroonScript binding

This chapter describes the DMeroonScript language and its binding with DMeroon. This language had been defined to test the DMeroon API and to write simple URLs to access DMeroon objects, it then evolved into a specific language. DMeroonScript is a tiny language that can easily be extended by the user (with DMeroonScript definitions) or by the implementor (in C). This language may be used to write simple single-line URLs to access DMeroon objects or quite complex multi-line programs to perform some actions on them. All the functionalities of DMeroon API may be put to work with the DMeroonScript language.

DMeroonScript is a stack-based, Forth-like language with a post-fix syntax. Constants are pushed onto the stack when processed. Other words correspond to commands whose definition is recorded in a linked list of vocabularies (instances of `UrlEnv`). Commands pop their arguments out of the stack and push their results. The stack only contains DMeroon objects.

DMeroonScript uses HTML to display its results so it is better to use it from an http client. The `DMeroon_interpret` API function also allows to submit DMeroonScript programs.

When an DMeroonScript is erroneous, its execution stops and an error message is returned (in html) and the content of the stack is displayed as well. When an DMeroonScript program stops normally, the stack should be empty; if the stack contains a single value, it is displayed. If the stack contains more than a single value, an error message is returned.

The empty DMeroonScript program triggers the display of an help message.

An DMeroonScript program is evaluated with a stack holding DMeroon objects and an environment mapping variables to DMeroon objects. An DMeroonScript is formed of a succession of constants or commands separated by spaces (or tabulations or newlines) or by slashes (this eases writing URLs).

5.1 Programs

A program is a possibly empty sequence of lines. Comments start with the hash sign and end up with the end of the line. Lines are made of tokens that is constants (i.e., literals) or commands. Commands are named as C identifiers, they start with a letter or an underscore and may contain a number of letters, underscores or digits. Succeeding commands that may be confused should be separated with whitespace, tabulations, newlines (LF or CR) or slashes.

Within DMeroon, programs are represented by an instance of class `String`. A NUL character of code zero ends the representation of a program even if it is not the last character.

5.2 Constants

Constants may be numbers, strings or characters. They are pushed onto the stack as soon as recognized.

Numbers are only positive. They may be written in decimal form or in octal form, if prefixed with `0` or in hexadecimal form, if prefixed with `0x` or `0X`. Numbers are converted into instances of the `Nat3` class.

Strings are enclosed with `"`. Characters within a string (and mainly `"` and `\`) may be protected by `\`. Strings may also be specified if surrounded by matching { and }. Quite often, this is used to denote an executable piece of code to submit to the `eval` command. Strings are converted into instances of the `String` class.

Characters are enclosed within `'`. They are converted into instances of the `Char1` class.

When created, a constant is initialized as copyable (unless the `DM_CONTEXT_DONT_INITIALIZE` Context option is set).

Examples

The following program pushes onto the stack an instance of `Char1`, two instances of `Strings` and three instances of `Nat3`:

```
'Q'
"This is a \"string\"."
        {This is { another } string}
12345 017 0xCafeBabe
```

5.3 Access to field

The content of the field of a DMeroon object may be extracted via the dot notation followed by the name of the field (specified directly and not as a string). The content of an indexed field may be extracted via the dot notation followed by the name of the field (specified directly and not as a string) suffixed by an index between square brackets. The result must always be a DMeroon object, this implies that the field must have type `reference`.

An error is signaled if an index appears and the field is not indexed. An error is signaled if the field is indexed and no index appears. An error is signaled if the field is secret.

> **Implementation note:** *Indexes are no longer implemented. In the past, a field such as a* `nat1`*, when read, was automatically reified into a* `Nat1` *instance. The reciprocal conversion was automatically done at writing-time. Should I reintroduce these features ?*

Examples

For instance, if the current site is on top of the stack then its `information` field is read by the `.information` command; this value replaces the site on top of the stack.

```
thesite                 # get the current site onto the stack
.information            # the DMeroon dictionary
.rest                   # get the 'user' dictionary
```

After removing all the previous comments, this little program may be equivalently written as:

```
/thesite/.information/.rest/
```

After the evaluation of either one of these two programs, the stack contains a single object: an instance of `Dictionary`.

5.4 Commands

A word that cannot be parsed as a constant or an access command (i.e., the name of a field prefixed with a dot) is a command whose definition has to be found in the current vocabulary. If the command is not defined, the DMeroonScript program is erroneous and stopped.

Commands are explained in this Section with the following notations: The stack is displayed with its top to the right, then the command is shown followed by the resulting stack (with still its top to the right). Uninteresting parts of the stack are elided with .... The class of the arguments is represented by the name of their class.

5.4.1 Basic commands

The basic vocabulary of the DMeroonScript language provides the commands that are used by the html display mechanism of DMeroon objects. These are: `thesite`, `byteprint`, `fancyprint` `entry` and `modify`.

... `thesite` → ... *Site* [DMeroonScript *commands*]
... *Object* `byteprint` → ...
... *Object* `fancyprint` → ...

The `thesite` command pushes the current site on top of the stack. The `fancyprint` command pops an object out of the stack and displays its content in html in a fancy form while `byteprint` displays its content in bytes (again in html form). These commands do not push anything in result. The `byteprint` command displays a result that is implementation-dependent. There is an additional `print` directive.

... *Nat3 Nat3 Nat3 Nat3* `entry` → ... *Entry* [DMeroonScript *command*]

The `entry` command pops four numbers out of the stack and pushes the `Entry` instance that has a key formed with these four numbers. Keys include a random part making the forgery of keys at least difficult.

... *String Object* `modify` → ... *Object* [DMeroonScript *command*]

The `modify` command allows to perform, on an object, a series of modifications encoded in a string. When an object has some mutable fields, these fields are displayed within an html form that when sent back to the DMeroon server with the *Modify!* button, triggers a POST request. A POST request is handled similarly to a GET request except that the content of the POST request, a string of *name*=*value*&... pairs, is turned into an instance of the String class and is pre-pushed in the stack before the evaluation of the DMeroonScript program. This command allows to modify many fields at once.

Implementation note: *This command knows how to modify all types of non-indexed fields but* `reference` *field.*

5.4.2 Vocabulary commands

The set of possible commands is held in a linked list of instances of `UrlEnv` i.e., vocabularies. When defined a command is recorded into the topmost vocabulary. It is not possible to redefined a command but it is possible to push a new empty vocabulary, to use another vocabulary or to pop the current vocabulary.

The evaluation starts with an empty vocabulary followed by the predefined vocabulary containing all the commands described in this chapter.

... `begin` → ... [DMeroonScript *commands*]
... `end` → ...
... *UrlEnv* `use` → *UrlEnv* ...

The `begin` command creates a new empty vocabulary and insert it in topmost position in the list of vocabularies. The `end` command reverses this effect and pops the current vocabulary. The `use` command allows to change entirely the linked list of vocabularies, it returns the previous vocabulary.

The current vocabulary may be obtained via the reifying `state` command.

... *Object String* `define` → ... [DMeroonScript *command*]

The `define` command binds a name and an object in the current environment. This implicitly creates a new command with that name that, when invoked, returns the associated object.

... *String String* `defcommand` → ... [DMeroonScript *command*]

The `defcommand` command binds a name and a string. When the command is invoked, the associated string is evaluated instead.

Examples

To be in an empty vocabulary with nothing at all defined is simple but problematic since there is no means to return to a regular situation: this is a dead-end:

```
"UrlEnv" 2 allocate use pop
```

The `print` command consumes the top of the stack. It is a simple matter to define a new command, named `idprint`, that avoids this behavior that is, that prints the top of the stack and leaves the stack unchanged. For instance,

```
{ dup print
} "idprint" define
1 idprint eval            # Test idprint
```

Rather than using `eval` explicitly, we may define an equivalent command and write:

```
{ dup print
} "idprint" defcommand
2 idprint                 # Another test
```

Here is a small DMeroonScript program that rolls the three objects on top of the stack. This is an extended `swap` command whose name is inspired from Forth similar command.

```
{ begin
    "top" define          # pop top and name it top
    "subtop" define       # pop subtop and name it subtop
    "subsubtop" define    # pop subsubtop and name it subsubtop
    subtop top subsubtop  # push all three of them in a different order
  end
} "roll" defcommand
"1" "2" "3" roll        #  Stack contains 1 3 2 (from top)
```

This example is safer than the two previous ones since the arguments of the `roll` command are created in a local vocabulary which is only used during the computation of the `roll` command. The name `top` may be used again safely.

5.4.3 Stack commands

These are Forth-like commands:

... *Object* `dup` → ... *Object* *Object* [DMEROONSCRIPT *commands*]
... $Object_1$ $Object_2$ `swap` → ... $Object_2$ $Object_1$
... *Object* `pop` → ...

These commands manage the stack. `dup` duplicates the top of the stack, `swap` swaps the two objects at the top of the stack while `pop` removes the top of the stack.

5.4.4 Boolean commands

These are the commands dealing with boolean and, chiefly, the alternative construction.

... `false` → ... *Boolean* [DMEROONSCRIPT *commands*]
... `true` → ... *Boolean*
... $Object_1$ $Object_2$ `==` → ... *Boolean*
... *Object* `not` → ... *Boolean*
... $Object_{else}$ $Object_{then}$ *Object* `if` → ... ...

These are the commands dealing with logical values. The `true`, resp. `false`, command pushes true, resp. false, onto the stack. True and false are predefined instance of classes `True` and `False`, subclasses of `Boolean`. The `==` command compares two objects and return the true or false boolean DMEROON objects. True is produced if the two compared objects are just the same physical object however, comparing two Nat3 numbers or two Strings yields true if they have same content. The `not` command returns true if given the boolean false and returns false for all other DMEROON objects. The `if` command pops out an object and two other objects known as *then* and *else*. If the object is false *else* is pushed otherwise *then* is pushed instead. Note that, in DMEROONSCRIPT, all values are equivalent to true but the false object which is the only one of its kind.

... `null` → ... *Object* [DMEROONSCRIPT *commands*]

The `null` command pushes `DM_NULL` i.e., the empty reference which belongs to any class.

Examples

This command allows to pop a stack until finding a `null` value. This `null` value is consumed. This function is erroneous if the stack does not contain a `null` value.

```
{ begin                               # Open a new vocabulary
    "x" define                        # Name the argument
    {pop-until-null} {} x null == if eval
  end                                 # Close the local vocabulary
} "pop-until-null" defcommand
{1} null 2 3 4 5 true pop-until-null    # Just leaves 1 on the stack
```

5.4.5 Arithmetic commands

The arithmetic operations exist as well. They are spelled in full letters and the only problem is to remember how not commutative operations take their arguments.

... *Nat3* *Nat3* `plus` → ... *Nat3* [DMeroonScript *commands*]
... *Nat3* *Nat3* `minus` → ... *Nat3*
... *Nat3* *Nat3* `times` → ... *Nat3*
... *Nat3* *Nat3* `quotient` → ... *Nat3*
... *Nat3* *Nat3* `modulo` → ... *Nat3*

There are also the other arithmetic comparators. Remember they are not commutative so pay attention to the order of the arguments. A quick rule is that you have to write as in a mirror.

... *Nat3* *Nat3* `le` → ... *Boolean* [DMeroonScript *commands*]
... *Nat3* *Nat3* `lt` → ... *Boolean*
... *Nat3* *Nat3* `ge` → ... *Boolean*
... *Nat3* *Nat3* `gt` → ... *Boolean*
... *Nat3* *Nat3* `ne` → ... *Boolean*

Examples

Here is the best-seller factorial example:

```
{ begin
    "n" define                                # Name the argument
    #"DEBUG: n= " print n print ", " print    # Print argument
    {n 1 n minus factorial times} 1 1 n le if eval
  end
} "factorial" defcommand
5 factorial print
```

5.4.6 Meta commands

There are two meta-commands. The `eval` command allows to evaluate a string while `state` reifies the current state of the DMeroonScript machine.

... *String* `eval` → ... ... [DMeroonScript *command*]

The `eval` command pops out a string and evaluates it. The evaluation of the string does not yield results per se, rather it may alter the stack and the current environment.

... `state` → ... *UrlState* [DMeroonScript *command*]

The `state` command reifies the full state of the evaluation into an instance of `UrlState`. Such an object appears in the message emitted in case of errors.

`... dumpstack → ...` [DMEROONSCRIPT *command*]
`...` *Nat3* `exit →`

To view the stack, one may print it with `dumpstack` (but you may alternately interactively inspect it with the `state` command). If you want to abort immediately the evaluation of a DMEROONSCRIPT program, use the `exit` command and tell it the error code you want to raise (or zero if this is an abrupt but normal end).

`...` *Site* `go → ...` *Site* [DMEROONSCRIPT *command*]

The `go` command allows to migrate computations from one site to another. The site where the computation should go is popped out of the stack, the current site is pushed instead and the whole state of the machine is copied and resumed on the intended site. The new site may return to the original site if using the top of its stack.

5.4.7 Miscellaneous commands

These are the left-over commands.

`...` *Object* `print → ...` [DMEROONSCRIPT *command*]

The `print` command pops an object out of the stack and tries to display its content into a short, comprehensible, human-readable form. Instances of the abstract classes `Value` (mainly for numbers) and `String` are recognized and printed specially.

`...` *String String* `extract → ...` *String* [DMEROONSCRIPT *command*]

The POST request pushes its body, a string of *name*=*value*&... pairs. The `extract` command allows to extract a value associated to a name from that string. The value is turned into an instance of `String`. This command may be used to extract the value of a given field from a string.

`...` *Entry* `fetch → ...` *Object* [DMEROONSCRIPT *command*]

The `fetch` command caches or updates locally the object associated to an Entry.

`...` *Clock* `tick → ...` [DMEROONSCRIPT *command*]

The `tick` command pops and increments a local Clock.

5.5 DMeroon API

There exist commands to invoke the functions of the DMeroon API. The arguments popped from the stack are in the order of the variables of the corresponding function of the API.

... *Object* `classof` → ... *Class* [DMeroonScript *commands*]
... *Object* `siteof` → ... *Site*
... *Object* `clockof` → ... *Clock*

The `classof` command pops an object out of the stack and pushes its class instead. Similarly the `siteof` command pushes the site that owns the object and `clockof` pushes the associated clock (that is, `NULL` if no clock is associated).

... *String* `class` → ... *Class* [DMeroonScript *command*]
... *String* `type` → ... *DMeroonType*
... *Class String* `field` → ... *Field*
... *String Nat3* `site` → ... *Site*

The `class` command pops a name and pushes back the predefined class with that name. An error is signalled if no predefined class with that name exists. The `type` command pops a name and pushes back the predefined DMeroon type with that name. An error is signalled if no predefined type with that name exists. The `field` command pops a name and a class and pushes the Field descriptor with that name from that class. An error is signalled if no field with that name exists. The `site` command pops an hostname and a port number and returns the appropriate DMeroon instance of `Site` if it exists.

... *Class Nat3...* `allocate` → ... *Object* [DMeroonScript *command*]
... *Object* `clone` → ... *Object*

The `allocate` command pops sizes (i.e., numbers) out of the stack until finding a class. It then allocates and pushes an instance of that class with the given sizes.

The `clone` command pops an object out of the stack, clones it and pushes the result back onto the stack.

... *Object* `setsharable` → ... [DMeroonScript *commands*]
... *Object* `setcopyable` → ...

These commands pops an object and makes it sharable or copyable.

... *Object Site* `setsiteof` → ... [DMeroonScript *commands*]
... *Object Clock* `setclockof` → ...

These commands allow to change the site owning an object or the clock monitoring an object.

... $String_{hostname}$ *Nat3* $String_{password}$ `connect` → ... *Site* [DMeroonScript *command*]

The `connect` command connects the current site with another site specified by the name of its host, its portnumber and the password to use.

... *Site Object* `send` → ... [DMeroonScript *commands*]
... `receive` → ... *Object Boolean*

The `send` command sends the reference of an object towards a given site. The `receive` command returns a boolean telling if an object has been received accompanied by the object if the boolean is true. When the boolean is false, the following object is meaningless.

... *Object Field* `xlenth` → ... *Nat3* [DMeroonScript *commands*]
... *Object Field* `get` → ... *Object*
... *Object Field Nat3* `xget` → ... *Object*
... *Object Field Object* `set` → ... *Object*
... *Object Field Nat3 Object* `xset` → ... *Object*

These commands allow to read the length of an indexed field or the content of a regular or indexed field or to modify a regular or indexed field. They are restricted to fields of type `reference`. You may modify fields with other type with the `modify` command.

... *Class String Nat3 Nat3* `smashclass` → ... *Class* [DMeroonScript *commands*]
... *Class String Nat3 String Nat3 Type Type Class* `createsubclass` → ... *Class*

The `createsubclass` command pops all the arguments needed to create a class. The `smashclass` command pops all the arguments needed to smash a class a number of levels.

5.5.1 Debugging command

The DMeroonScript language allows to debug applications through HTTP and unsafe additional commands.

... *Nat3* `address` → ... *Object* [DMeroonScript *command*]

The `address` command pops out a number and pushes the DMeroon object whose DMeroon reference is equal to this address. The word `address` is misleading since the number is the DMeroon reference not the address of the object viewed from the binding language.

... *Object* `proxyof` → ... *External* [DMeroonScript *command*]

This command returns the proxy of a DMeroon object. It must only be used for debugging since these objects are normally hidden from the user.

... *Nat3* `setdebuglevel` → ... *Nat3* [DMeroonScript *command*]

This command allows to change the debug level of the DMeroon server. Zero means terse, higher numbers increase verbosity. This command returns the previous debug level. This command is meaningless if the server is compiled without the `cpp` variable `DM_DEBUG` defined.

5.6 Use

DMeroonScript is used to inspect DMeroon values with an http client. If your DMeroon server runs on your local machine then the initial URL is:

`http://localhost:56423/`

This URL gives access to some general information and provides a link to allow the inspection of the site object. This URL is:

`http://localhost:56423/thesite/fancyprint`

From this, one can follow references that correspond to URLs looking like:

`http://localhost:56423/0x000a67dc/0x6353e384/0x3448a335/0x1f91cd56/entry.object`

This kind of URL specifies an object externalized with a given key.

When an object is displayed (see figure 5.1), a banner tells which DMeroon server was the source of the answer. The object is preceded by a sentence specifying its main characteristics such as immutable or mutable, local or remote, sharable or copyable or uninitialized, obsolete or up-to-date; its class is also mentioned and it is possible to see the bytes representing the object if you are an hex fan[1]. The fields of the object are then displayed in order starting with the name of the field, its type, its length if indexed and followed by its content. If a field is mutable, then it is possible to modify it directly in your http client and press the *Modify!* button (see figure 5.2).

5.7 DMeroonScript API

It is possible to run DMeroonScript programs from the binding language. The DMeroonScript language may easily be extended in C with new commands. Here follows the API to extend, in C, the DMeroonScript language.

The DMeroonScript machine has a state, an instance of the `UrlState` class.

[1] *des sixties, comme disait Serge Gainsbourg.*

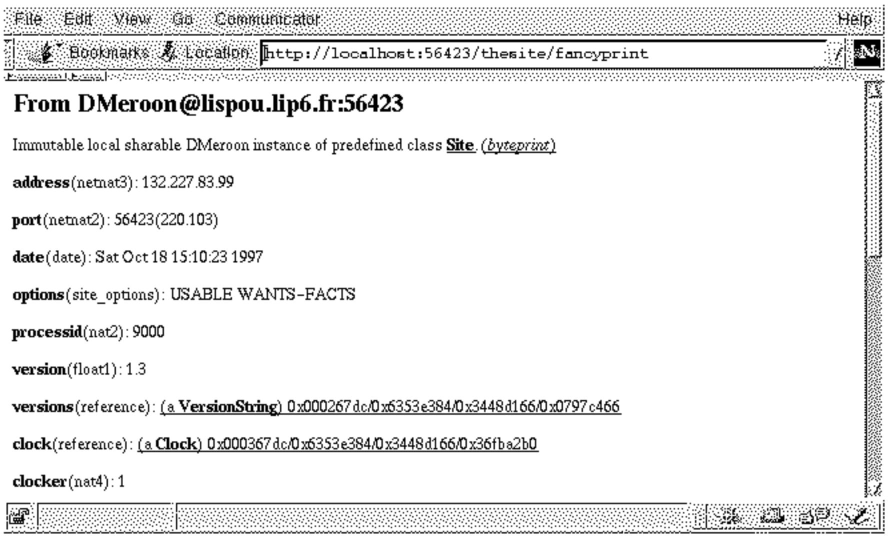

Figure 5.1: A Site displayed in html

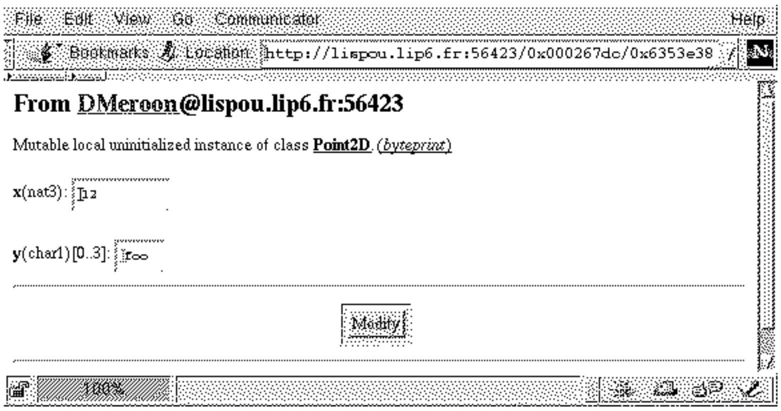

Figure 5.2: A mutable object displayed in html

`UrlState` [*Class*]

`previous`	*another UrlState*
`script`	*the* DMEROONSCRIPT *program (a* `String`*)*
`pc`	*the program counter (a* `nat3` *number)*
`stack`	*the stack (a* `BoundedStack`*)*
`environment`	*the vocabulary of commands (an* `UrlEnv`*)*
`errcode`	*the last error code*

This class defines the various components of the state of the DMEROONSCRIPT machine. Objects are pushed onto the stack and the stack is extended when needed. The program counter is an index into the script (a `String`). To invoke the `eval` command in the DMEROONSCRIPT creates a new state for this evaluation. When the evaluation is over, the old state is restaured. Commands are looked for in the environment, a linked list of hash-tables containing instances of `UrlActions`.

`UrlAction` [*Class*]

`behavior`	*a pointer to a C function*
`internal`	*a* DMEROON *object or NULL*
`name`	*a* `nat1`*-sequence of* `char1`
`argument`	*a* `nat1`*-sequence of* `Class`*es*

An `UrlAction` records an DMEROONSCRIPT command. It has a name, is implemented by a C function with a precise signature. The arguments of that command must be DMEROON objects only and must have the class specified in the indexed `argument` field. The `internal` field holds any DMEROON value that may be needed by the command. This field is used, for instance, by the `define` and `defcommand` DMEROONSCRIPT command and may also be used to mimic closures.

The `UrlAction` class may, of course, be subclassed.

The C function that implements an `UrlAction` command must have the following prototype:

[*UrlAction behavior*]

```
DM_UrlState* behavior (
        DM_Context   *context,
        DM_UrlAction *self,
        DM_UrlState  *state,
        void         *output_port,
        DM_Object    *arg0,         /* references only */
        ... )
```

This *behavior* function receives the current context, the `UrlAction` describing it (so it may fetch the `internal` value for instance), the state of the DMEROONSCRIPT machine (with its stack for instance), the output port where to print (or `NULL` if no printing is needed) and as many variables as there are arguments. The DMEROONSCRIPT machine takes care of invoking behaviors with the correct number and classes of arguments. A command may take from 0 to 8 arguments. A command return the new state of the machine.

The only difficulty is to remember how are mapped arguments from the stack to variables of the behavior: they are mapped as represented in the previous definitions. If a command has three arguments, the last one is at the top of the stack (this is to simulate how stacks are displayed in this chapter).

Example

Here is the code of the setdebuglevel command:

```
static DM_UrlState*
DM_setdebuglevel_command (DM_Context   *context,
                          DM_UrlAction *self,
                          DM_UrlState  *state,
                          void         *output_port,
                          DM_Nat3      *arg0 )
{
  DM_Nat3 *result = DM_Allocate0(context, Nat3);

  result->value = DM_TheCurrentSite()->debug_level;
  if ( context && DM_HasNotOption(context, DM_CONTEXT_DONT_INITIALIZE) ) {
    DM_SetCopyable(context, result);
  }
  DM_TheCurrentSite()->debug_level = arg0->value;
  DM_push_object_in_bounded_stack(context,
                                  (DM_Object*)(result),
                                  &(state->stack) );
  state->errcode = DME_SUCCESS;
  return state;
}
```

5.7.1 Explicit evaluation

To run an DMeroonScript program is simple. Just give a C string containing the DMeroonScript program and a number of objects that will form the initial stack (first object is the first pushed onto the stack). These objects must be DMeroon objects, they are prefixed by their number. The final state of the DMeroonScript machine is returned so the results may be popped out of the final stack. The DMeroonScript machine is run in mute mode: nothing is printed.

```
DM_ErrorCode                                                    [Level 0]
DMeroon_interpret (DM_Context*  context,
                   DM_UrlState* *urlstate,
                   char*        urlprogram,
                   DM_nat3      nobjects,
                   DM_Object*   object, ... )
```

The previous function is implemented on top of the following one which is less usable as it stands but more accurate since it represents the current DMeroonScript interpreter.

```
DM_UrlState*                                                    [Level 0]
DM_interpret_url_script (DM_Context*  context,
                         DM_UrlState* urlstate,
                         void*        output_port /* or NULL */ )
```

This function runs an DMeroonScript machine with an initial state. When the DMeroonScript machine terminates, its final state may be accessed to discover the objects that are pushed in its stack. The interpretation stops as soon as an error occurs. Printing may occur via the port mentioned as third argument; `NULL` prevents printing.

Examples

It is sometimes interesting to combine two accesses but not to force the intermediate object to be fetched on the current site. Here is a way to achieve that:

```
errcode = DM_site_of(context, &other_site, o);
/* Check error code */
errcode = DM_interpret(context, &urlstate,
                       "go swap .field1 .field2 swap go",
                       2, o, other_site );
/* Check error code */
... urlstate->stack->item[0] /* is the result. */ ...
```

The computation is migrated onto the owning site of the `o` object, `o.field1` is made present on that site and `o.field1.field2` is returned onto the original site.

5.8 Development environment

You may use the *TOP*`/DMeroon/Commands/urlscript` command to submit clearer, longer, multi-lined DMeroonScript programs from any shell command interpreter.

`urlscript` *[hostname[:port]] [file]* [*Command*]

This command establishes a connection to the specified DMeroon server and sends it the specified file using the DMeroonScript protocol. If no file is mentioned, the standard input is accumulated (until an end-of-file) before being sent. Shell variables are substituted with their values before being sent to the server. The server answers with html which is delivered onto the standard output of the `urlscript` command. The final errorcode returned by the DMeroonScript program becomes the return code of the command. Files containing DMeroonScript programs have `.dms` as extension.

An excellent tool, named `dmClient`, allows you to really easily edit and submit DMeroonScript requests to a DMeroon server. Jean-Michel Inglebert is the author of `dmClient`. `dmClient` may be operated from any JavaScript-enabled http client. Point your favorite browser to *TOP*`/Contrib/dmClient/dmClient.htm` page to play with.

There also exists an Emacs mode to edit DMeroonScript programs. Two variables may be set to specify the DMeroon server you want to send the content of the current buffer. The result is viewed with Emacs thanks to the marvellous `w3` package from William Perry[2]. This Emacs mode appears in the *TOP*`/Emacs/`DMeroonScript`.el` file. Documentation appears in its header.

5.9 Examples

This is a one-liner to increment the general clock of a site. It displays the string “Attention” followed by the current clock, increments the clock and redisplays it (in a fancy way i.e., with a “tick” button). The script ends by printing “Done.” Here it is:

```
"Attention"/print/thesite.clock/dup/tick/fancyprint/"Done."/print
```

[2] ftp://ftp.cs.indiana.edu/pub/elisp/w3

This is a bigger example of an DMEROONSCRIPT program that records the string `DMeroon is great!` in a dictionary named `queinnec` which is grafted in the user dictionary.

```
# Get the ‘‘user’’ dictionary.
thesite.information.rest "user_dictionary" define

# Create a new dictionary (with the right size).
"Dictionary" class 9 allocate "new_dictionary" define

# Initialize the ‘‘name’’ field.
"name=queinnec" new_dictionary modify pop

# Initialize the ‘‘value’’ and ‘‘rest’’ fields.
new_dictionary dup classof "value" field "DMeroon is great!" set pop
new_dictionary dup classof "rest" field user_dictionary.rest set pop

# Finish the initialization and makes the dictionary sharable.
new_dictionary setsharable

# Hook the new dictionary after the ‘‘user’’ dictionary.
user_dictionary dup classof "rest" field new_dictionary set pop

# Report correct execution.
"Done" print
```

Chapter 6

ICSLAS binding

This chapter describes a binding initiated some years ago with a concurrent, distributed extension to the Scheme language: the ICSLAS language. The ICSLAS language is powered by a byte-code interpreter, a compiler exists (written in Scheme) for the Scheme subset of ICSLAS. The byte-code machine and the compiler are enhanced versions from the ones described in [Que96].

All ICSLAS values are represented as DMEROON objects. Entities necessary for the byte-code machine such as lexical environments, global environments and continuations are as well represented by DMEROON objects thus allowing their migration.

Implementation note: *Unfortunately, I had no time to pursue that effort since 1996.*

Bibliography

[AF94] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[Coi87] Pierre Cointe. The ObjVlisp kernel: a reflexive architecture to define a uniform object oriented system. In P. Maes and D. Nardi, editors, *Workshop on MetaLevel Architectures and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[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.

[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.

[QC88] Christian Queinnec and Pierre Cointe. An open-ended Data Representation Model for Eu-Lisp. In *LFP '88 – ACM Symposium on Lisp and Functional Programming*, pages 298–308, Snowbird (Utah, USA), 1988.

[Que90] Christian Queinnec. A Framework for Data Aggregates. In Pierre Cointe, Philippe Gautron, and Christian Queinnec, editors, *Actes des JFLA 90 – Journées Francophones des Langages Applicatifs*, pages 21–32, La Rochelle (France), January 1990. Revue Bigre+Globule 69.

[Que91] Christian Queinnec. Meroon: A small, efficient and enhanced object system. Technical Report LIX.RR.92.14, École Polytechnique, Palaiseau Cedex, France, November 1991.

[Que93] Christian Queinnec. Designing meroon v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), September 1993.

[Que94a] 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.

[Que94b] Christian Queinnec. Sharing mutable objects and controlling groups of tasks in a concurrent and distributed language. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of the Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, Lecture Notes in Computer Science 907, pages 70–93, Sendai (Japan), November 1994. Springer-Verlag.

[Que96] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[Que97a] Christian Queinnec. Distributed generic functions. In Jean-Paul Bahsoun, Takanobu Baba, and Jean-Pierre Briot, editors, *Second France-Japan workshop on Object-Based Parallel and Distributed Computing — OBPDC '97*, Toulouse (France), October 1997.

[Que97b] 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.

[WS90] Larry Wall and Randal L Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1990.

Appendix A

DMeroon source files

This chapter gives some insight on the sources of DMeroon, this might be useful *(i)* when adapting DMeroon to another language, *(ii)* when porting to another Operating System or language, *(iii)* for curious people.

There are many possible dimensions for customization or modularization of DMeroon, Figure A.1 try to place the various files with respect to layers, modules and relationship between them.

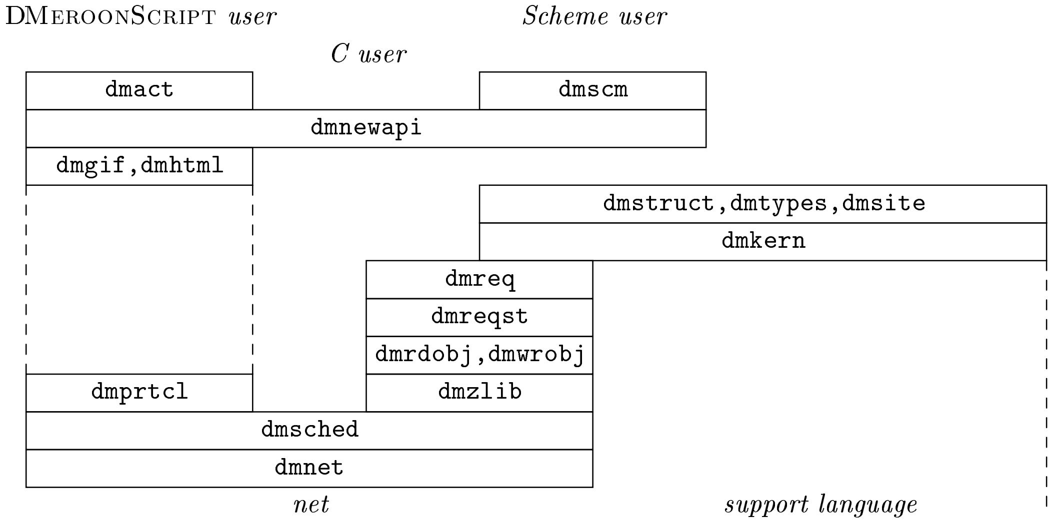

Figure A.1: Layers and relationship between files

A.1 Structures of files

The top directory of the distribution is named `icslas` (perhaps suffixed with a version number). Let *TOP* be this directory. Under *TOP* may be found various `README` files and other administrative files.

When you rebuild the DMeroon library for a given language, the compilation takes place in another directory, the *TARGET* directory. Once DMeroon is rebuilt for a given language, the obtained library should not be used with another language. It is nevertheless possible to rebuild DMeroon concurrently for all languages for which a binding exists.

The *TOP* directory contains a number of sub-directories.

- The `DMeroon` directory contains the files that are independent of any binding languages. This directory also contains a number of sub-directories.

  - The `c` directory contains the C files.
  - The `Commands` contains shell or Perl scripts.
  - The `Doc` directory contains the current documentation and some other related research papers.
  - The `Test` directory contains DMEROONSCRIPT test programs as well as other tests written in Scheme.

- The `C` directory defines C as a binding language and so is the `Bigloo` directory. Former bindings were studied but are no longer maintained, these are `OScheme`, `Pico`. `XXX` contains template for bindings.
- The `Bootstrap` isdirectory used by the maintainers of DMEROON to regenerate some files belonging to DMEROON distribution (i.e., tarballs).
- The `Commands` directory contains (shell or Perl) scripts and so is `DMeroon/Commands`.
- The `Others` directory contains Scheme code from other authors while `Library` contains Scheme code belonging to the ICSLAS runtime library.
- The `config` directory populated with various Imakefiles. These parts are combined using `imake` to form the complete Makefile in the *TARGET* directory. This process is pretty complex and plenty of bells and whistles may parameterize it.
- The `Contrib` directory contains DMEROON-fans contributed code.

The *TOP* directory also contains some interesting files such as — the top `Imakefile` (which was carefully written to be usable as a regular Makefile), — the `ChangeLog` file of the ICSLAS project, — various `README` files.

A.1.1 The *TOP*/DMeroon/c/ directory

The DMEROON library is made of a number of C files populating the *TOP*/`DMeroon/c` directory. Each C file has an associated `.h` file defining the prototypes of the exported variables or functions but, to allow an easy use of the DMEROON system, you only have to include the `dmeroon.h` header file which, in turn, includes all the specialized header files such as `dmstruct.h`, `dmkern.h`, etc. The `dm.h` header file is also automatically included. This file is a little peculiar since it defines macros that are used in all these other header files. The `dmeroon.h` header file also recursively includes two specialized header files. The first header, `dmos.h`, contains all the dependencies with respect to the Operating System (it tries to automatically discover the operating system).

The second header file (also automatically included) describes the peculiarities of the binding language: this is *TOP*/`Bigloo/dmbgl.h` for Bigloo, *TOP*/`C/dmc.h` for C, *TOP*/`OScheme/dmoscheme.h` for OScheme, etc. When DMEROON is bound with some language, it requires access to specific routines of the runtime of this language to help DMEROON manage its space or comply with the conventions of data representations so that DMEROON objects look like normal objects. This header file is often complemented by some code adding to the runtime of DMEROON. For instance, for C, DMEROON uses Boehm's Garbage Collector.

Names are rather systematic and I hope easily decipherable. Except for the `DM_` prefix they all have, variables and functions have lower-case names (made of `underscore_separated_words`). Functional macros names are made of `JointTogetherCapitalizedWords` while data macro names are written, as usual, in `UPPER_CASE`.

Most of the data structures required by DMEROON are statically allocated so loading DMEROON is a snap. All these files end with an initializing routine for the initializations that cannot be static. When compiled with the appropriate debug level, many checks are performed in these initialization routines.

Nearly all functions have a `Context` instance as first argument, even internal functions. This allows to confine errors or to parameterize certain behaviors.

Documentation may be automatically extracted from these files to form html pages that may be browsed by your favorite html client. To get them, run `make regenerate.documentation` in the *TARGET* directory. The html pages will appear in *TARGET*/`WWW`. and the

Basic files

TOP/DMeroon/c/dmstruct.dm — This important file defines all the C structures that are used by DMEROON. All these structures are also DMEROON classes and so may be inspected as regular DMEROON objects thus satisfying introspective reflexivity. This file has a Scheme syntax (in fact a MEROON syntax) and is automatically turned into a `dmstruct.c` and a `dmstruct.h` files with help of the `dm2ch` compiler.

TOP/DMeroon/c/dmeroon.h — This is the header file that recursively includes what is necessary to compile C files using DMEROON resources. Not very big by itself but rather complex.

TOP/DMeroon/c/dm.h — An enormous header file, far less readable than `dmeroon.h` and far more complex (only for die-hards). It defines many of the internal macros that are used throughout the code of DMEROON.

TOP/DMeroon/c/dmos.h — This file contains the dependencies with respect to Operating Systems. Currently these are all UN*X-based and appropriate for SUN4 bsd, SUN4 solaris, DEC Alpha, DEC Ultrix, Sony News, PC Linux. The selection of the right OS is automatic but I plan to use `autoconf` some time.

TOP/DMeroon/c/dmerr.h — This header file defines the zillions error codes detected by DMEROON accompanied with some explanations.

TOP/DMeroon/c/dmid.h — The current version of DMEROON.

TOP/DMeroon/c/dmsizes.h — This file defines how DMEROON types are mapped onto C types (i.e., `nat3` is often an `unsigned int`). The current mapping is universal with respect to the previously mentioned Operating Systems but you may have to change it if DMEROON detects a misfit. DMEROON checks compatibility when initialized (see `dmtypes.c` initialization) and, in case of disagreement, displays the appropriate size and alignment constraints to help you. Some examples of sizes and alignments are kept in the *TOP*/DMeroon/c/HINTS file.

Core files

From now on, the remaining parts are composed of a `.h` and `.c` files. They are classed in alphabetical order.

TOP/DMeroon/c/dmact.c — This file defines the DMEROONSCRIPT language.

TOP/DMeroon/c/dmcache.c — This file defines the utilitary functions to manage caches when marshaling/unmarshaling objects.

TOP/DMeroon/c/dmcomm.c — This file defines the management of Entry and Exit items that support the remote pointer mechanism.

TOP/DMeroon/c/dmdbg.c — This file defines some debug utilities for DMEROON. They are rather useful when porting DMEROON but not after.

TOP/DMeroon/c/dmextr.c — This file defines extra libraries for users' comfort (in C).

TOP/DMeroon/c/dmgif.c — This file defines some icons.

TOP/DMeroon/c/dmhash.c — This file defines the basic machinery for hashtables. Pay attention, searching is done via a macro, not by a function (for typing reason).

TOP/DMeroon/c/dmhtml.c — This file defines the HTML server which allow to inspect DMEROON instances from any http clients.

TOP/DMeroon/c/dmkern.c — This contains the functions that constitutes the basic DMEROON memory model i.e., how to access fields within an object, whether an object is local or not, initialized or not, how to increment a clock, to clone an object, to create new classes. This file also contains some predefined DMEROON constant objects that are used throughout the implementation such as nil, true, false, uninitialized. These constants and functions are not part of the API.

TOP/DMeroon/c/dmlib.c — This file is there to hold functions lacking from some Operating Systems (you know the kind of things such as `strstr`, bad `memcpy` etc.). It is also used to mask some differences such as `putenv` versus `setenv` that cannot be resolved in `dmos.h`.

TOP/DMeroon/c/dmnet.c — This file defines the TCP/IP related parts of DMEROON. It allows to listen for connections, to fill/flush buffers, to send/receive messages.

TOP/DMeroon/c/dmnewapi.c — This file defines the API for the C language.

TOP/DMeroon/c/dmprint.c — This file defines a series of C functions to print every predefined DMEROON type. This is mainly used for debug and html generation.

TOP/DMeroon/c/dmprtcl.c — This file defines how protocols used on TCP connections are recognized and handled.

TOP/DMeroon/c/dmrdobj.c — This file defines how to decode sequences of bytes into DMeroon objects over TCP connections. This is viewed as a kind of bytecode interpreter implementing a stack-oriented machine.

TOP/DMeroon/c/dmreq.c — This file manages requests queues and their answers.

TOP/DMeroon/c/dmreqst.c — This file defines functions that manage distribution at low level. It checks for locality, handle communication requests.

TOP/DMeroon/c/dmsched.c — This file defines the DMeroon event loop. It actually implements various variations. DMeroon may be run in a unique process, in a thread, or from Scheme (where threads are encoded as continuations).

TOP/DMeroon/c/dmscm.c — This is the generic interface for Scheme. It translates the C API into a set of functions more amenable to be used from a Scheme implementation.

TOP/DMeroon/c/dmsite.c — This file defines the original Site object which is the root of DMeroon self-description.

TOP/DMeroon/c/dmtypes.c — This file checks whether DMeroon types are well mapped onto C types, it also dynamically associates DMeroon types to the appropriate C read/write functions.

TOP/DMeroon/c/dmwrobj.c — This file defines how to encode DMeroon objects over TCP connections. This encoding is viewed as a compilation for the decoding machine. As for compilers, many different encodings are possible, that's why this file has been more heavily changed than the dmrdobj.c file.

A.2 Rebuilding DMeroon

Rebuilding DMeroon is performed in the *TARGET* directory. First, this *TARGET* directory is created then, it is populated with a lot of files: a generated Makefile, some config files memorizing pathnames in various formats (shell, makefile or Scheme). Appropriate files from *TOP* are copied. Whenever possible, files are linked rather than copied. Eventually make world is performed in the *TARGET* directory and should yield a fully operational DMeroon server for a given language.

Running make from the *TOP* directory will ask interactively some questions. The first question is what to build: answer d for DMeroon. The second question concerns the binding language: answer c for C or b for Bigloo. The script will try to identify the sort of machine it is running on, will create and populate the *TARGET* directory then will propose to rebuild DMeroon. Answer y then and wait.

A.3 Other files

Other files are required to adapt DMeroon to a particular language. The binding is detailed in Chapter A.4.

A.3.1 C

The binding with C requires some additional files that are located in the *TOP*/C directory. These are:

TOP/C/dmc.h — *TOP*/C/dmc.c — These are the binding files. They explain how to map DMeroon values in C values. DMeroon needs a GC, two are possible: Boehm's GC or no GC at all (with the latter, the server will crash after saturating the swap space).

TOP/C/server.h — *TOP*/C/server.c — This file implements a very simple DMeroon server. It has several options, described in table A.1, that may be specified as arguments to the command.

There is also a *TOP*/config/c.mkf file that describes how to regenerate DMeroon for C. This file also contains some stand-alone tests.

A.3.2 Bigloo

The binding with Bigloo requires some additional files that are located in the *TOP*/Bigloo directory. These are:

TOP/Bigloo/dmbgl.h — *TOP*/Bigloo/dmbgl.c — These are the binding files. They explain how to map DMeroon values in Bigloo values.

`Option`	Meaning
-banner	Prints a banner.
-verbose	displays some information.
-linger *duration*	The server will finish after *duration* seconds of inactivity. Used as last option, the duration may be elided to mean infinity.

Table A.1: Options of the rudimentary C server

TOP`/Bigloo/dm.bgl` — This file (written in Scheme) is a Bigloo module defining the DMeroon server above Bigloo.

TOP`/Bigloo/main*.bgl` — These are the main Bigloo modules that start the server.

TOP`/Bigloo/dmwriter.h` — *TOP*`/Bigloo/dmports.h` — These are files helping to patch files of the distribution of Bigloo.

There is also the *TOP*`/config/bigloo.mkf` file that describes how to regenerate and how to test DMeroon for Bigloo.

A.4 Binding DMeroon

DMeroon is currently bound to C and to Bigloo (a Scheme dialect). It was bound to OScheme (another dialect of Scheme) and Pico (still another dialect of Scheme). These last bindings are to be updated. It is envisioned to bind DMeroon with Caml, C++, Emacs, Java, Linux kernel, Tcl, etc.

The sources of DMeroon are parameterized with respect to the support language i.e., the language with which it is bound. In effect, DMeroon statically allocates all its predefined classes, fields, types as well as the site object and therefore needs to statically know the format of the objects as imposed by the binding language. DMeroon objects look like objects of the binding language that is, DMeroon objects are embedded into native objects of the support language.

DMeroon also has to manage its memory space i.e., allocate and free objects, it uses the GC of the support language otherwise DMeroon brings its own GC. All these parameters are specified by macros and functions that are required to compile DMeroon sources. This means that a DMeroon library compiled for one support language can probably not be used by other support languages.

To bind DMeroon with some *xxx* language is supported by at least three files.

- *TOP*`/Xxx/dm`*xxx*`.h` containing macros and prototypes,
- *TOP*`/Xxx/dm`*xxx*`.c` containing runtime functions or objects,
- *TOP*`/config/`*xxx*`.mkf` is the Makefile to compile the port.

These three files are often built altogether since they are highly inter-related. You may take a look at these files for Bigloo, C or Icslas.

A.4.1 The `dm`*xxx*`.h` header

First prevent double inclusion to save `cpp` time. Then include any header file that are required for the *xxx* support language.

Define `DM_SUPPORT_LANGUAGE` to be a string with the name of the support language, something like `"`*xxx*`"`. This string will appear in the `support` field of the site object as a `String` instance.

`DM_MALLOC` should be the name of the routine that allocates objects for the binding language. As C's `malloc`, this routine takes the number of bytes to allocate and returns a pointer onto the allocated object. See the `DM_allocate` function in the `dm`*xxx*`.c` file below.

The difficult part is how to embed DMeroon objects within *xxx* objects. Often objects in *xxx* have a particular header prefixing the content (with respect to *xxx*) of the object. This content consists of the

DMEROON header followed by the content (with respect to DMEROON) of the associated fields, see figure 2.1. It is important not to confuse the three views of the same object, an object may be referenced by a pointer in *xxx*, another one in DMEROON, and a third one if considered by `DM_malloc`.

Figure A.2 shows that, in C, pointers designate the first byte of the content of the object. Class and proxy are accessed backwards from this pointer. To be able to recognize DMEROON objects, we use a tag and to respect alignment constraint some padding is added. The low-level `DM_MALLOC` returns a pointer on the first byte of the object i.e., its tag.

Bigloo objects are handled via the address of the Bigloo tag that is , the address returned by `DM_MALLOC`. Some padding is then added for alignment constraint. DMEROON can be recognized with a specific Bigloo tag.

Icslas just uses DMEROON representation and adds nothing to it.

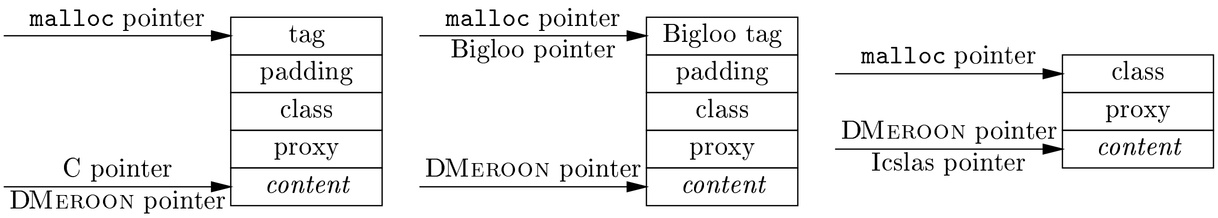

Figure A.2: Objects in C, Bigloo and Icslas

Viewed from `DM_MALLOC`, the prefix of a DMEROON object is described by the following universal structure, taken from the *TOP*`/DMeroon/c/dm.h` file:

```
typedef struct DMeroon_object {
  DM_SupportPrefix()
  DM_DMeroonPrefix();
} DMeroon_object;
```

You must therefore define, from the `DM_MALLOC` point of view, the `DM_SupportPrefix` macro describing the structure of the object and the `DM_SupportPrefixInitialization` macro that initializes this structure. This latter macro receives the name and the exact type of the object to be initialized so the initialization may use this information (to record the size of the object for instance).

You must then define how to recognize a DMEROON instance among the values handled by the *xxx* support language. You may also define this macro to be always true if you like to live dangerously. This is the `DM_IsDMeroonInstance` macro which takes a pointer, viewed from *xxx*, onto an *xxx* value, and which returns a C boolean.

Another macro performs the same task, `DM_IsObject`, but this one takes a DMEROON reference (instead of a pointer onto an *xxx* value) and returns a C boolean. Since a `NULL` value is a legal DMEROON pointer, this macro should take care of this fact.

To be able to switch from the DMEROON world to the *xxx* world and back, two fundamental macros are `DM_ToSupport` that takes a DMEROON reference and returns an *xxx* pointer, and `DM_FromSupport` that takes an *xxx* pointer and returns a DMEROON reference. In the source files of DMEROON, a variable prefixed by `wrapped_` holds a value of the support language.

The `DM_SupportPointer` macro defines the type of a pointer onto an *xxx* value. A pointer onto a DMEROON object has the `DM_Object*` type. A pointer onto an object viewed from `DM_MALLOC` has type `void*`. Don't confuse them!

The glue to bind DMEROON may require some initialization code, the function to call is mentioned by the `DM_INITIALIZE` macro.

Finally, the `DM_RaiseException` macro must be defined, it receives an errorcode and has to take care of handling that situation. It may signal a recoverable exception if the binding language supports it, it may `longjmp` in C to a safer context, it may print the error code and aborts the process.

This is the end of the minimal binding.

The `dmscm` layer

If you bind DMEROON to a language with runtime types such as Scheme or maybe Java, then you may use the `dmscm` layer that insulates you more from gory details. This layer is a new API above the one described in chapter 3 offering facilities more like the ones of chapter 4. Values of the support language are automatically converted into their appropriate representation in DMEROON back and forth. You'll have to define additional macros describing this situation.

A.4.2 The dm*xxx*.c file

This file contains the runtime functions that are necessary to glue DMEROON with the *xxx* support language. Some functions must be defined since they are required by DMEROON: this is the case of the `DM_allocate_bare_instance`, the `DM_print_char_star` and the `DM_INITIALIZE` functions.

All allocations performed by DMEROON are done via a single function named `DM_allocate_bare_instance`. This function has the following prototype:

```
DM_Object*
DM_allocate_bare_instance (DM_Context *context,
                           DM_Class   *class,
                           DM_nat3    size )
```

Given a class and a size in bytes, it should allocate an *xxx* value able to contain a DMEROON object of a given size (in bytes). The function should return a DMEROON reference onto the freshly allocated object (or abort if the allocation cannot be performed). The object must be correctly initialized with respect to the *xxx* support language. It will be initialized with respect to DMEROON point of view by other DMEROON routines shortly after. Pay attention to alignment, the first byte referenced by the DMEROON pointer should respect the most stringent alignment constraint imposed by the hardware or compiler[1]. DMEROON performs some sanity checks to detect such things.

All values are printed by DMEROON using the `DM_print_char_star` function. Its prototype is:

```
void
DM_print_char_star (DM_Context*        context,
                    char*              string,
                    DM_nat3            len,
                    DM_SupportPointer  wrapped_port)
```

The second argument is a C string to print, the third if not zero is the number of characters to print out of the string. If the third argument is zero then the whole C string is printed up to its end i.e., up to a NUL character. The fourth argument is an *xxx* value representing the output stream where to print. Depending on the *xxx* language, this argument may be very different things: a C stream, a C file descriptor, a Scheme port, etc.

The last function that must be present is the function mentioned by the `DM_INITIALIZE` macro. This function, often named `DM_`*xxx*`_initialize`, performs some checks and initializes the binding.

A.4.3 The *xxx*.mkf file

This file is the Imakefile describing how to regenerate DMEROON with the *xxx* support language. Most often the *xxx*`.mkf` file complements the `all.mkf` file which defines how to regenerate a generic DMEROON executable.

[1] On my home machine, `gcc` aligns long doubles on 8-bytes boundaries while the hardware only requires 4-bytes alignment.

A.5 Writing code for DMeroon

If you ever want to add code to DMeroon, you must follow some rules. This set is far from complete but will be enriched as I remember them.

- Indent your code (I personally use the default indent style of Emacs). Don't use long lines i.e., bigger than 78 characters.
- Try to be portable. Check portability on at least another computer system.
- Mark temporary lines with a TEMP comment; mark future code to be written with a FUTURE comment (accompanied by a short description).
- Every bug, if corrected, must lead to a comment and to a test in some test suite.
- Try to use the facilities of the autodoc utility (that builds html pages from source files).
- Wrap test or sanity code within `DM_DebugAtLevel` directives.
- Compile with all the possible warning options and never leave a warning appear. The `-Wall` option for `gcc` is recommended.
- Declare external functions in *file*`.h` and define them in *file*`.c`. Add these files to the global `Makefile` architecture.
- Be very cautious about allocations:
  - don't use mutable global data structures, try to be thread-safe.
  - don't call `malloc`, use DMeroon objects instead,
  - better, try to stack allocate these DMeroon objects,
  - don't let bounded data structures limit your algorithms.
- Respect naming conventions for identifiers. Names are rather systematic and I hope easily decipherable. Except for the `DM_` prefix they all have, variables and functions have lower-case names (made of `underscore_separated_words`). Functional macros names are made of `JointTogetherCapitalizedWords` while data macro names are written, as usual, in `UPPER_CASE`.
- DMeroon puts a high emphasis on self-description and parameterization. Use DMeroon objects for that goal.

Appendix B

Object internal representations

This appendix describes the various implementation states DMEROON objects may have. This is a rather technical chapter but is of great help for me.

Viewed from a site, a DMEROON object may be *local* or *remote*. Local means that the current site owns it. A remote object is always associated with an Exit item. Again viewed from a site, an object is either *present* (i.e., local or remote but cached) or *absent* (that is remote and not cached). Moreover when a remote object is present, it may be *validly cached* or *obsolete* i.e., its cache is up to date or out of date.

Viewed from users, the objects they have pointers on are always present. They never can access proxies i.e., entry or exit items.

Some constants are used to encode these states. Constants in DMEROON are instances of the `Constant` class, a direct subclass of `Object`.

B.1 Local object

When an object is allocated, `DMeroon_allocate` returns it in the uninitialized state. The class of an allocated object is always present and so is its owning site since this is the current site. An object stays in this state until being initialized, semi-externalized or externalized, see the state diagram in Figure B.1 (some uninteresting states are omitted from this Figure).

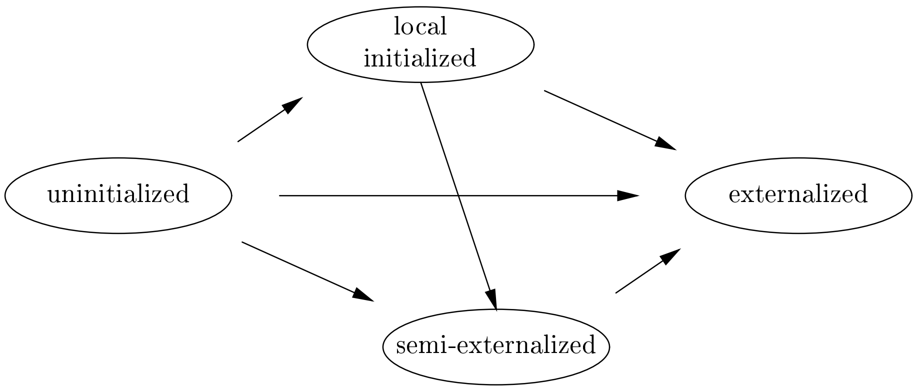

Figure B.1: States of local objects

Implementation note: *The constants that are used to mark these states have historical names and will probably be changed. These representations were choosen to lessen memory consumption: a local object costs two additional references (and maybe some padding) compared to C (which does not require any overhead). This cost increases when objects are externalized.*

The class field of the DMeroon header always contains a reference to a present class which the object is a direct instance of.

B.1.1 Uninitialized

A local uninitialized object has as proxy the `uninitialized` constant, see figure B.2. Allocation returns DMeroon objects in this state.

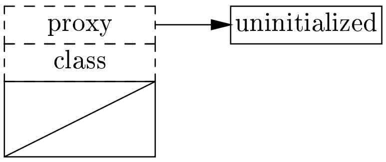

Figure B.2: Local uninitialized object

B.1.2 Local sharable

When an unitialized object is declared sharable, then its proxy becomes the `nil` constant.

B.1.3 Local copyable

When an uninitialized object is declared copyable, then its proxy becomes the `true` constant.

B.1.4 Semi-externalized

If a local object is to be monitored by a clock (via `DM_set_clock_of`) then it must be semi-externalized (since the clock can only be held in a field of an Entry item), see figure B.3 (in this figure, the dotted squares shows the details of the fields of the Entry item). To be semi-externalized means that the object has an Entry item as proxy. In this state, the object is still unknown from remote sites. To be semi-externalized has no relation with its initialized state: the object may still be declared initialized, sharable or copyable (the difference is that these properties are now recorded as options in the associated entry item).

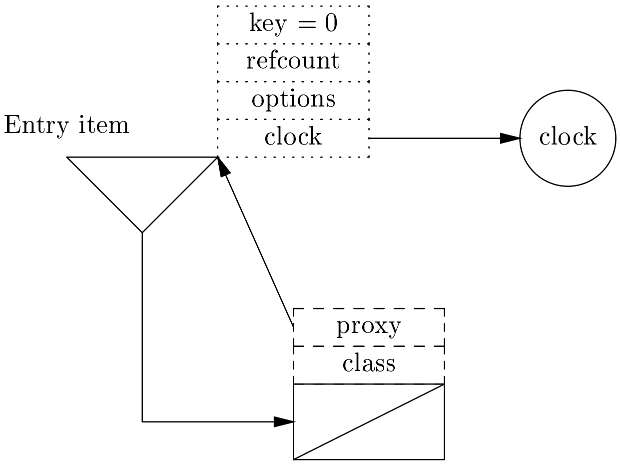

Figure B.3: Semi-externalized object

Entry items are, for the implementation, regular DMeroon objects, instances of the `Entry` class defined as follows. An externalized or semi-externalized object has an instance of `Entry` as proxy, this proxy points back to the (semi-)externalized object via the `object` field. That is, The class of a semi-externalized object is always semi-externalized or externalized. The owning site of a semi-externalized object is always semi-externalized or externalized.

`Entry` [*Class*]

`number`	*a unique* `key` *identifying the object*
`counter`	*a reference counter (for GC)*
`options`	*some options describing the Entry item*
`object`	*the object the entry item stands for*
`clock`	*its associated* `Clock` *if any*

When an object is semi-externalized, an Entry item is allocated and becomes the proxy of the object. The key (i.e., the number under which the object will be known from the network) of the Entry item is null but the `clock` field is set to hold the monitoring clock. The Entry item is itself an immotile DMeroon object which is in the uninitialized state and will be left in that state all its lifetime. Recall that Entry items are never seen by the user.

If the object was copyable before becoming semi-externalized then the `options` field of the `Entry` item has the `DM_ENTRY_IS_COPYABLE` option set. If the object was sharable instead before becoming semi-externalized then the same option is unset. If the object was uninitialized then the previous option has no meaning but the `DM_ENTRY_IS_UNINITIALIZED` option is set instead.

If a semi-externalized object is made copyable then the `DM_ENTRY_IS_UNINITIALIZED` option is unset and the `DM_ENTRY_IS_COPYABLE` option is set. If a semi-externalized object is made sharable then the `DM_ENTRY_IS_UNINITIALIZED` option is unset and the `DM_ENTRY_IS_COPYABLE` option is unset. A semi-externalized object is uninitialized when the `DM_ENTRY_IS_UNINITIALIZED` option of its Entry item is set.

A direct instance of `Entry` with a null key corresponds to a semi-externalized object.

B.1.5 Externalized

An object is externalized when it needs to be known from other sites either because it is reachable from another object displayed in html for an http client (that may want to follow the link i.e., the reference), or because a reference onto the object was sent to another site. When externalized, an object has, as proxy, an Entry item with the same set of options as in the semi-externalized state. Additionally the Entry item *(i)* gets a non null key, *(ii)* is registered in the hashtable holding all `Entry` items, *(iii)* its reference counter is initialized with zero.

When externalized, a mutable object must be monitored by a Clock. By default and if not explicitly specified, this Clock is the general Clock of the current site (held in the `clock` field of the site). If the object is immutable then the `clock` field is set to `DM_NULL`. An object may be externalized without being semi-externalized first and without even being initialized.

When an object is externalized, its class is also externalized as well as its owning site.

Any time a reference onto a DMeroon object is sent, its reference counter is incremented; however the reference counter is not incremented when a html link is sent. When a site detects that it does not need any longer a reference, it discards it and sends a decrement message to the associated reference counter. When the reference counter reaches zero, the Entry item may be reclaimed since no DMeroon site needs its associated object. After the Entry item is reclaimed, the object itself may be reclaimed if locally useless.

When an object is externalized, a key is generated for it. The key is unique and will never be reused anywhere, anytime (the key is based on the IP and port numbers of the server, the current date, a regularly incremented counter and two random bytes to make the forgery of keys at least difficult). When an object is migrated i.e., is owned by a different site than its birth site, it keeps its key.

A direct instance of `Entry` with a non null key corresponds to an externalized object. The reference counter may be zero or strictly positive. Zero means that no site knows it, however the object had been externalized for the sake of some http client.

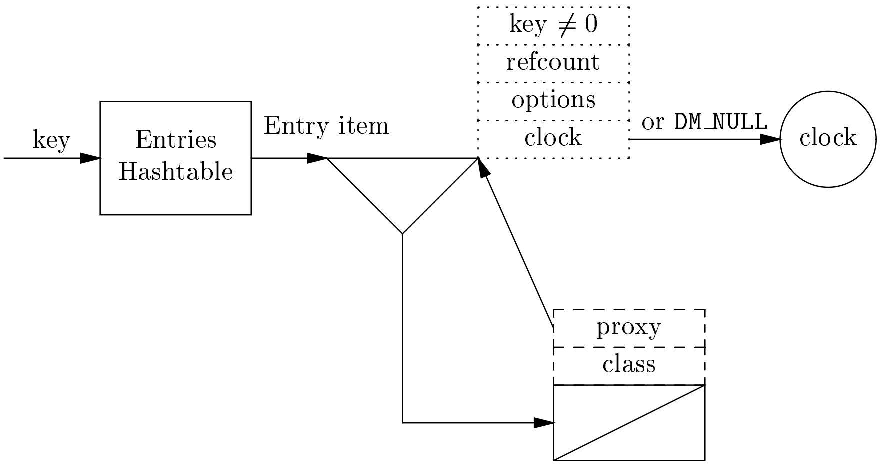

Figure B.4: Externalized object

B.2 Remote

Only the unmarshaling routines can create remote references. An Exit item is a big data structure containing enough information to fetch the content of the remote object, to know its associated class, clock or site, to perform non-local GC. When a remote object is present i.e., locally cached, it looks like a local object except that its proxy is an Exit item. Intermediate states exist to cope with dynamically created classes whose instances are sent before their class.

When an object is remote, it is only remotely referenced, as in figure 2.2, by an Exit item. Remember that outside of the API, users never see Exit items. When a reference onto an object is sent, the object is externalized on its owning site and an Exit item is created on the receiving site, see Figure B.5 for the most complex case.

The `Exit` class inherits from the `Entry` class thus an Exit item may be externalized (as an Entry item) and be known from other sites. The `Exit` class is defined as follows:

`Exit` [*Class*]

`number`	*a unique* `key` *identifying the original object*
`counter`	*a reference counter (for GC)*
`options`	*some options describing the Exit item*
`object`	`DM_NULL` *or a replica*
`clock`	*the associated* `Clock` *if any or* `DM_NULL` *(if immutable) or an* `Exit` *if absent*
`class`	*the class of the remote object or an* `Exit` *if absent*
`time`	*the time when the replica was fetched*
`from`	*the site from which we got the Exit item (maybe unusable)*
`owner`	*the site owning the remote object (maybe unusable)*
`next`	*used internally*

When a site unmarshals an Exit item, it tries to pre-fetch asynchronously the class and the clock of the remote object if they were unmarshaled as remote pointers. Therefore the `class` field is either a pointer to a present instance of `Class` or a pointer to an Exit item leading to a remote class. Once the class is present, the current site tries to pre-fetch asynchronously its clock. Therefore the `clock` field is either `DM_NULL` if the object is immutable, a present instance of `Clock` or a pointer to an Exit item leading to a remote clock. The content of the `site` field is always an instance of `Site`, this site is always registered in the hashtable of known sites (but the site may be in the unusable state).

Only when the content of the `class` and `clock` fields are present, the remote object can be fetched to

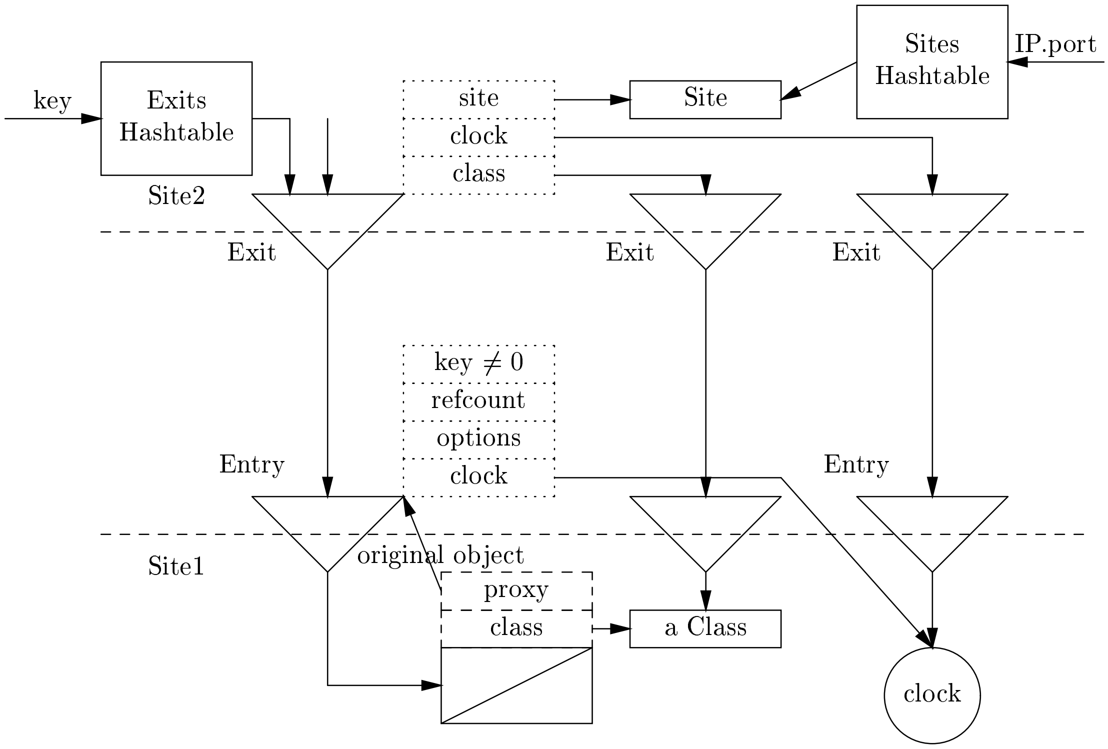

Figure B.5: Exit item with absent class and absent clock

form a local replica of the remote object. This replica is reachable via the `object` field of the `Exit` item and has as proxy the Exit item. DMEROON tries to shortcut pointers to the Exit item into direct pointers to the replica, see figure 2.3.

DMEROON may, for GC reasons, reverts a pointer to a replica into a pointer to the associated Exit item and then reclaims the replica. To fetch again the replica will be automatically redone if needed. If an Exit item has a locally cached replica then this replica has this Exit item as proxy.

When a replica is present, its class and clock are always present. The converse may not be true.

B.2.1 Reexported

An Exit item may be marshaled towards a site in which case, the part of the Exit item which is inherited from the `Entry` class is filled (except the key which is already set): the reference counter is initialized to zero (it counts the number of times the Exit item is reexported [Piq91]) but the `object` field of the Exit item may be null since a replica does not need to be present nor its class or clock. See figure B.6.

B.3 Site

Sites are handled particularly. All sites objects are reachable from a hashtable of locally known sites. Sites are lazily propagated between DMEROON servers. A site instance may be usable or not, but its fundamental fields (i.e., IP number, port and date of creation) are always up to date. A usable site is a site whose content may be read, for that, a usable site must be validly cached and has , as proxy, an Exit item (whose `owner` field is that very site). A site without Exit item as proxy is not usable. However normal users (outside the API) never see that internal state. A special option `DM_SITE_IS_USABLE` identifies that state.

Only the unmarshaling command `DM_SITE_COMMAND` can create instances of `Site`, an uninstantiable class. Such a site is not initialized and only its fundamental fields are correct i.e., its `address`, `port`, `date` fields and additionally, perhaps, the `route` field. It is possible to send objects to sites for which no Connection is

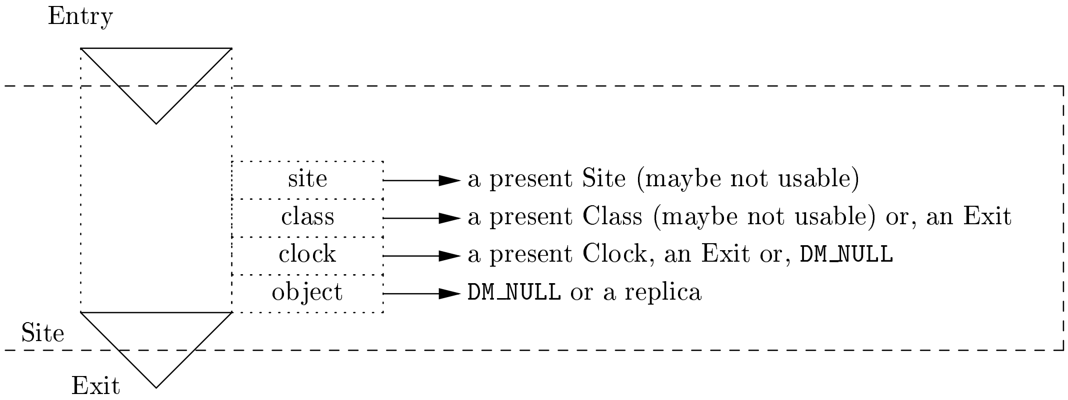

Figure B.6: Reexported Exit item

open. The `route` field refers to a third site from which the current site heard of the site and which acts as a communication relay. It is also possible to open a more direct Connection. When a site is usable, or it has a opened Connection in its `connection` field or it has a relay site in its `route` field.

B.4 Class

For a class to be usable, it must be present, its superclasses must be usable and all its fields must be present. Users outside the API cannot perceive that state. This state is identified by the `DM_CLASS_IS_USABLE` option; this option is reset when a class is transmitted. A site that receives a class tries to prefetch missing components to ensure its usability.

B.5 Miscellaneous

Out of the API, users only have pointers onto present (and usable for classes and sites) objects. They never see Entry or Exit items. These objects may be out of date but to read them will force DMeroon to refresh them.

Inside the API, a reference may be to a present object or to an Exit item. The Exit item may be associated to a replica or not, this replica may be valid or not. Before a replica may be brought locally, the class of the object must be present, as well as its clock if mutable. The fields of Entry or Exit item should be considered with great care since classes or sites may be unusable.

B.6 Properties

These data structures are complex and respect some invariants. This Section is intended to be useful for implementors, it describes the same representational invariants bottom-up i.e., starting from the implementation.

We will use the following terminology. A proxy is an entry or exit item. An object is a regular DMeroon value that is not a proxy. Pay attention that exit items are, from the implementational standpoint, indirect instances of **Entry**; however when we say that something is an entry item, we exclude the exit item case.

The user of the DMeroon API can only see present objects. An implementor may see objects or proxies. Objects and proxies are always handled via pointers so when we say an object, we imply a reference to an object.

B.6.1 Object

An object is always present, its class is always present. Its proxy may be:

- the `uninitialized` constant meaning that the object is uninitialized,
- the `nil` constant meaning that the object is uninitialized but declared sharable,
- the `true` constant meaning that the object is uninitialized but declared copyable,
- an exit item meaning that the object is remote but locally cached. The object then appears in the `object` field of the exit item. The exit item is registered in the exits hashtable of the current site, the `class` field of the exit item refers to a present class (?). The `site` field of the exit item contains a present site.
- an entry item meaning that the object is either:
  - semi-externalized if the key of its entry item is zero then the entry item is not registered in the entries hashtable of the current site. If a clock had been explicitly associated to the object then it is held in the `clock` field of the entry item.
  - externalized if the key of its entry item is non zero. The entry item is registered in the entries hashtable, its reference counter tells how many times the object was marshaled to other sites.

  In both cases, the class is also at least semi-externalized and so is the current site. The object may still be uninitialized or sharable or copyable: this is specified by bits in the option field of the entry item.

Two special cases exist for classes and sites.

Class

A remote but present class may have non present fields. The DMeroon library tries to fetch as soon as possible remote fields of present classes. When the class has the `USABLE` option set then its fields are all present. The user may only see usable classes and therefore can only allocate objects with usable classes.

Site

The current site is a regular object, remote sites are special. When a site heard of a remote site, an instance of `Site` is allocated and filled with the basic information (IP, port number) and stored in the sites table of the current site. If it needs to, the DMeroon library fetches the content of the site and sets its `USABLE` option on. The user may only see usable sites.

B.6.2 Entry item

An entry item (a direct instance of the `Entry` class) always has an object in its `object` field which in turn has the entry item in its `proxy` field. The class of the object and its owning site (the current site) are present and externalized. The key of the entry item may be zero if the object is only semi-externalized or non-zero if the object is externalized. In this latter case, the entry item appears in the entries hashtable of the current site.

The object the entry item stands for may be uninitialized or sharable or copyable. If the object is sharable then it is monitored by a clock (held in the `clock` field of the entry item. The object may be uninitialized and already associated to a clock.

The object may be known from other sites if externalized. A reference counter of zero is a sure sign that the object was displayed in html for some http client. An entry item with a null reference counter and associated to no clock or to the general clock of the site may be recycled by the GC.

B.6.3 Exit item

An exit item stands for a remote object. An exit item appears in the exits hashtable of the current site. The `site` field of the exit item refers to a site which is present but not necessarily usable.

The object may be cached or not. If the remote object is cached then its `object` field holds a present object i.e., a replica of the remote object. The replica has the same structural information the remote object has: same class, same length for indexed fields. In this case, the `class` field of the exit item refers to a present class.

The exit item may be re-exported in which case, it is registered as well in the entries hashtable of the current site. The exit item may be locally cached or not.

If the remote object is not cached, the `object` field of the exit item is NULL, the `class` field may be a present class or an exit item leading to a remote class.

Appendix C

Protocols

When started, DMEROON acts as a server, listening to a port and waiting for incoming connections. When a connection is opened, the first characters are scanned to determine which protocol they obey. Currently, a DMEROON server recognizes *(i)* the `GET` and `POST` requests of the HTTP protocol, *(ii)* the DMEROONSCRIPT protocol (extending the two first protocols), *(iii)* the `DMeroon` protocol which is used to exchange binary-encoded DMEROON requests between DMEROON sites.

This chapter describes protocols and shows how DMEROON servers may be extended to handle new protocols. It requires some understanding of DMEROON implementation and predefined classes as they appear in the *TOP*`/DMeroon/c/dmstruct.dm` file. Warning! not all details are present here, read the sources and chiefly the *TOP*`/DMeroon/c/dmprtcl.c` file!

The DMEROON implementation uses only DMEROON objects, this allows to debug parts of DMEROON with DMEROON itself. When DMEROON is initialized, it normally listens to a TCP port whose port number is specified by the environment variable `DMEROON_SERVING_PORT`. If this variable is not set, then the value of the `DM_DMEROON_FAVORITE_PORT` cpp variable is used that is, the very well-known port 56423. If `DMEROON_SERVING_PORT` is the zero number then no Server is created and DMEROON will not serve any requests. In the numerous cases where a server is created, an instance of the `Server` class reifies this situation.

When a connection is detected, an instance of the `Connection` class is created. This instance contains two buffers to hold bytes to emit or bytes received via the connection. The buffer for incoming bytes is an instance of the `InBuffer` class, the other is an instance of the `OutBuffer` class.

When enough characters are received to form a line, i.e., a sequence of bytes terminated by a newline character, then the first characters are scanned to determine which protocol may be used to handle the connection. To this end, a connection instance has a field (the "`protocol`" field) initially filled with the equivalent field of the current site object. This field is a linked list of instances of the `Protocol` class. The `Protocol` class is defined hereafter:

`Protocol`		[*Class*]
`next`	*next protocol to try*	
`handler`	*a C function*	
`name`	*a* `nat1`*-indexed field of* `char1`	

A protocol is defined with a name (the sequence of characters that triggers this protocol) and a handler (a pointer to a local C function to handle the connection). Each protocol name is tried in turn and compared against the first incoming characters. If no protocol is recognized, the connection is immediately closed. When a protocol is recognized, the associated handler is invoked, with the current context, on the incoming buffer.

When a protocol is recognized, it is its duty to consume the incoming characters. The associated handler is stored in the "`handler`" field of the connection. This is used for instance by the `DMeroon` protocol to handle new incoming requests. The handler returns a C boolean telling whether the request has been handled or not. Therefore, to return `DM_FALSE` means that, while the first line (with the letters determining the protocol) is present, the rest of the request is still not present. In this case, the whole process will be retried later when the connection receives additional characters: this allows a protocol to wait until the entire request is present in an `InBuffer`. It is up to the protocol to decide when the request is complete. For instance, DMeroon prefixes messages with their lengths while the `SchemeScript` protocol for Bigloo waits for a sequence of NUL NUL CR LF. While buffers are bounded, DMeroon ensures that the whole request is held in a single instance of the `InBuffer` class.

When the handler returns `DM_TRUE`, it must have consumed the characters of the request, it thus has to modify the '`index`' field of the incoming buffer.

The order of protocols is significant since they are tried in turn. Of course, a protocol triggered by a null string catches everything.

C.1 Existing protocols

Besides the three main protocols that is, HTTP, DMeroonScript and DMeroon, there also exist other small specialized protocols. They are currently useless but serve as examples.

The `#` protocol swallows empty lines or lines starting with a sharp sign or a white space. This protocol does not answer anything but after the line is swallowed, it starts again to determine which protocol to use to handle the remaining characters. This is the comment protocol.

The `help` protocol just answers the identification of the DMeroon site then closes the connection. One may write `?` instead of `help`.

A third protocol is the `error` which prints the rest of the line on the standard error output of DMeroon. It does not close the connection.

The `GET` protocol answers http requests in html form. The produced html pages may trigger a `POST` protocol to mutate objects. The `%dd` convention may be used for these two protocols. It allows to hide characters under their ASCII code.

The `GET` protocol is also used to serve the `/robots.txt` file (to prevent compliant robots to explore a DMeroon server) and the `/icons/dmeroon.gif` icon files (to illuminate the html pages generated by the DMeroon server).

The `DMeroon` protocol is a byte-oriented communication protocol that allows DMeroon servers to exchange values. See the paper (in French) called "*sérialisation—désérialisation en* DMeroon" for details. After entering the `DMeroon` protocol, the connection is used in inner DMeroon protocol, see chapter C.4.

The Scheme binding with Bigloo offers another protocol, the `SchemeScript` protocol which waits for a string containing an Sexpression to evaluate. The result is sent back to the client that sent the expression then the connection is closed. The expression is evaluated with Bigloo's `eval` function in the global shared environment. This is a dangerous but powerful protocol defined in the *TOP*`/Bigloo/dmbgl.c` file that can be used with the *TOP*`/DMeroon/Commands/schemescript.prl` command.

The use of the last protocols may be restricted with a password. This password may be specified at runtime by the `DMEROON_PASSWORD` environment variable or at compilation time by the `DMEROON_ADDITIONAL_PASSWORD` cpp variable. The principle is very simple, the name of the protocol is suffixed with this string of characters. Passwords are currently limited to 64 characters by the `DMEROON_ADDITIONAL_PASSWORD_LENGTH` cpp variable. The password is immediately concatenated to the name of the protocol: pay attention to leading spaces.

> **Implementation note:** *Reading the* `pgp` *man page, I realized how unsafe is the* `DMEROON_PASSWORD` *environment variable. This will be improved in some future. Uses of passwords make, at least difficult, connections to* DMeroon *servers protected by different passwords.*

C.2 Adding protocols

New protocols may be added dynamically. You clearly are an expert if you use this function!

```
DM_Protocol*                                                    [Internal]
DM_add_protocol (DM_Context*  context,
                 char*        string,
                 DM_c_boolean (*handler)(DM_Context*, DM_InBuffer*) )
```

This primitive defines a new protocol to be recognized by the current site. All new connections will know this new protocol but already opened ones will not. This protocol will be tried first so it can mask existing protocols with longer names but same prefix (you may alternatively directly hack the 'protocol' field of the current site). The handler must return the C boolean DM_TRUE if it consumes the request or DM_FALSE if the request is not entirely present. At that time, when the connection receives additional characters then DMeroon will call again the handler function. It is possible for the handler to extract from the InBuffer instance, the associated Port, OutBuffer and other related values.

When the handler is called, the 'index' field of the InBuffer designates the first character to be handled. Since the handler is called, the first line is present in the buffer i.e., there is a LF character somewhere after this index. The buffer may be used up to the 'maximum' field which designates the first character which is not part of the buffer.

If you want to register a protocol that must be protected by the password of the DMeroon server then, you have to use this function instead:

```
DM_Protocol*                                                    [Internal]
DM_add_restricted_protocol
                (DM_Context*  context,
                 char*        string,
                 DM_c_boolean (*handler)(DM_Context*, DM_InBuffer*) )
```

C.3 Some classes

There are many classes used in the message layer. Two sites may exchange message through a Connection. Except for a short time while a Connection is being initialized, the site field may be not usable.

`Connection` [*Class*]

`number`	the portnumber or file descriptor
`options`	options
`...`	TCP stuff
`last_read`	last date an `InBuffer` was read
`last_write`	last date an `OutBuffer` was flushed
`reception`	the associated `InBuffer`
`emission`	the current associated `OutBuffer`
`emitqueue`	the first `OutBuffer` to flush
`clocker`	the last time, the buffer was flushed
`site`	the site at the other end of the connection
`protocol`	the list of protocols that may be used
`handler`	the function to handle the protocol
`password`	the password to use

The most important parts of a Connection are the buffers to consume and produce. When a message arrives, it is always recorded in a single `InBuffer`.

`InBuffer` [*Class*]

`port`	the associated Connection
`cache`	a `BoundedStack` acting as a stack/cache
`index`	the next character to read
`maximum`	the first character not to read
`amount`	the total number of characters acquired by the Connection
`byte`	the characters

When marshaling an object, the `OutBuffer` may be full, a new `OutBuffer` is then allocated and linked in the `next` field. Since there is the possibility to abort a message while composing it, multiple `OutBuffer`s may stand in memory.

`OutBuffer` [*Class*]

`port`	the associated Connection
`cache`	a `BoundedStack` acting as a stack/cache
`mark`	mark the beginning of a message
`index`	the next character to write
`maximum`	the first character not to write
`amount`	the total number of characters flushed by the Connection
`next`	the next `outBuffer`
`byte`	the characters

C.4 The DMeroon inner protocol

This section describes the DMeroon protocol i.e., the language used by DMeroon servers to serialize or deserialize objects. The serialization is viewed as a compilation into bytecodes towards the deserializer wich acts as a bytecode interpreter. There is usually more than one way to serialize some data, the serializer may choose to be eager (and pre-send i.e., push values) or to be lazy (and let the receiver pull the missing values). The deserializer is the interpreter of a value-oriented stack-based prefix language. The stack may as well be managed as a cache. Each Connection has separate associated caches for reading and writing.

The DMeroon protocol is triggered by the `DMeroon` word, then the connection is turned into the inner DMeroon protocol. In this inner protocol, messages are exchanged. A message corresponds to a sequence of bytes that will be deserialized into a single object. The content of `reference` fields may also be pre-sent.

Commands are encoded by a single byte and may be followed by arguments. In addition to the commands described below, the commands corresponding to the characters CR, LF and SP are ignored. Comments are also skipped: they start with a sharp character and end with the first following CR or LF. These conventions allows to comment a byte-stream, for instance, to use the `#!` convention of Unix.

There are a number of invariants to respect when marshaling objects: the most important one is that it is not possible to unmarshal an object if its class is not usable withing the receiver. This is why, when sending objects, only a remote pointer is created. Before asking for the content of an object, the receiver ask for its class (and also its clock). Objects are then pulled rather than pushed. However you may pre-send these objects if you are sure that the receiver knows their class or if you use the `LIMIT` command.

The following notations are used below: *object*, *clock*, *class* etc. means that an instance of that class is expected. DMeroon types such as *nat1*, or *class-options* may also appear.

C.4.1 Ubiquitous objects

These commands return ubiquitous objects i.e., objects that are local to every site (even if differently implemented).

`NIL` [*unmarshaling commands*]
`TRUE`
`FALSE`
`UNINITIALIZED`
`NULL`

The first niladic commands that is, `NIL`, `TRUE`, `FALSE`, `UNINITIALIZED` and `NULL`, just transmits a simple ubiquitous object. Ubiquitous objects are mainly used by the implementation (or by the Scheme binding); although they exist on every site they are considered equal. The site of an ubiquitous object as obtained with `DM-site-of` is always the current site. `NULL` allows to transmit the null reference i.e., a reference that does not lead to any DMeroon object.

`PREDEFINED-CLASS` *nat1* [*unmarshaling commands*]
`PREDEFINED-TYPE` *nat1*
`PREDEFINED-FIELD` *nat1 nat1*

The `PREDEFINED-CLASS` command takes a *nat1* as argument and returns the predefined class with that index among the set of predefined classes. Such a class is an ubiquitous object even if its representation varies from site to site. The `TYPE` command takes a *nat1* and returns the DMeroon type with that index (see table 2.1). Such a type is an ubiquitous object even if its representation varies from site to site. The `PREDEFINED-FIELD` i j command returns the ith field of the jth predefined class.

C.4.2 Stack-related commands

These commands deal with the stack, they may take an additional argument. The stack is a regular stack if managed from the top, it is considered as a cache if managed from the bottom. Technically, a Tcp

Connection between two sites is associated to four stack/caches — two per communication way. When the emitter changes its stack/cache, it encodes the command that will modify accordingly the stack/cache of the receiver.

Implementation note: *I may separate some time stacks from caches.*

`POP` [*unmarshaling commands*]
`SWAP` *object*
`DUP` *object*
`RESET` *object*
`PUSH` *object*

The commands `POP`, `SWAP`, `DUP` and `RESET` manage the stack/cache of the deserializer. First, there is a stack which is accessed with the usual functions. It is possible to extract or replace values from this stack with an index counted from the top of the stack or from the bottom of the cache. `RESET` totally clears the cache. `SWAP` swaps the top of the stack with the element right under it, the subtop of the stack (of course, the stack must have at least two elements). `DUP` duplicates the top of the stack i.e., it pushes the object on the top of the stack in the stack (of course, the stack must contain at least one object). The three previous commands do not return a DMEROON object *per se*, they just perform the required action then read the object that follows and return it.

`POP` returns the object standing on the top of the stack as well as it removes it from the stack. `PUSH` reads its argument, pushes it onto the stack, then it returns that object as value. Stacks are automatically extended whenever needed.

`REFER-WITH-NAT1` *nat1* [*unmarshaling commands*]
`REFER-WITH-NAT2` *nat2*
`REFER-WITH-NAT3` *nat3*

The commmand `REFER-WITH-NAT1` (resp. `REFER-WITH-NAT2` or `REFER-WITH-NAT3`) takes a *nat1* (resp. a *nat2* or *nat3*) argument and returns the object cached at that position counted from the bottom of the cache. The cache must be large enough for this index to be legal (in particular less than the stack pointer).

`INSERT-WITH-NAT1` *nat1 object* [*unmarshaling commands*]
`INSERT-WITH-NAT2` *nat2 object*
`INSERT-WITH-NAT3` *nat3 object*

These commands take an index and an object, insert the object in the cache at the given position (counted from the bottom of the cache) and return this object. The cache must be large enough for this index to be legal (in particular less than the stack pointer).

C.4.3 Useful commands

`EMITTING-SITE` [*unmarshaling commands*]
`RECEIVING-SITE`

Commands `EMITTING-SITE` and `RECEIVING-SITE` are shortcuts that encode the two sites standing at both ends of a TCP connection. These sites are known after a brief handshake that takes place immediately after the Connection is opened. This handshake or bootstrap of the Connection has to be specially done since it must warm up the inner DMEROON protocol and proceed to the exchange of site objects.

`SET-COPYABLE` *object* [*unmarshaling commands*]
`SET-SHARABLE` *object*

`SET-COPYABLE` (resp. `SET-SHARABLE`) takes a DMEROON object as argument and initializes it to be copyable (resp. sharable), it returns that initialized object. These commands are often needed after the `ALLOCATE` command.

`CLASS` *class-options class* [*unmarshaling commands*]
`FIELD` *field*
`SYMBOL` *symbol*

`CLASS` takes some class-options and a class as arguments, it returns this very class. Non-predefined classes must be prefixed by this command to be usable on a remote site. A class must be usable before you may use it to allocate instances. A class is usable when all its super-classes and all its fields are present and usable.

`FIELD` takes a field as argument and returns it. Non-predefined fields must be prefixed by this command to be usable on a remote site.

`SYMBOL` takes a symbol as argument, interns it in the local hashtable of symbols (so two symbols with the same name are the same), and returns it as value.

`RECEIVE-REQUEST` *request* [*unmarshaling commands*]
`RECEIVE-ANSWER` *answer*

The two commands `RECEIVE-REQUEST` and `RECEIVE-ANSWER` take a DMEROON object as argument and return it as value. `RECEIVE-REQUEST` (resp. `RECEIVE-ANSWER`) enqueues the object which must be an instance of `Request` (resp. `Answer`) in the list of pending requests (resp. answers). These requests or answers will be handled after the deserialization of the current message.

> **Implementation note:** *This is to avoid simple deadlocks when a site has to unmarshal a message and requires, for it, another message from the same site.*

`PROG1` *object object* [*unmarshaling commands*]
`PROG2` *object object*

The PROG1 (resp. PROG2) command takes two objects as arguments and returns the first (resp. the second) of them.

LIMIT *nat3 object* [*unmarshaling command*]

This command returns its second argument but ensures that its encoding is exactly *nat3* bytes (this size includes the representation of the *nat3* number itself up to the last byte of the object). Moreover, if an error occurs while unmarshaling the object, the error is caught and NULL is returned instead. This command is often used to pre-send values whose classes may be unknown from the receiver.

C.4.4 Specific commands

These are commands that create objects, remote pointers, etc.

ALLOCATE *class sizes...* [*unmarshaling command*]

This command takes as first argument a class followed by *nat** sizes. It allocates and returns an instance of the class whose indexed fields are determined by the given sizes. For instance, to allocate a String of 3 characters, you may write (8 is supposed to be the index of the String predefined class):

```
ALLOCATE (PREDEFINED-CLASS 8) 3
```

The parentheses that appear above are cosmetic only, they group commands to ease reading.

The class that appears as first argument must be usable. Note that it is not required to specify the number of sizes since this may be deduced from the class.

FILL *object content...* [*unmarshaling command*]

This command takes an object as argument, fills it with the serialized content and finally returns it. The content is made of the concatenation of the serialized fields. For instance, to fill the previously mentioned String of three characters, one may write:

```
FILL (ALLOCATE (PREDEFINED-CLASS 8) 3) F o o
```

The parentheses that appear above are cosmetic only, they group commands to ease reading.

EMITTER-REFERENCE *key external-options class clock* [*unmarshaling command*]

The EMITTER-REFERENCE command creates a remote pointer onto an object owned by the emitting site. The key is the number that names the associated Entry item or a reexported Exit item. The class may be encoded as a remote pointer itself, the receiving site will fetch it in order to be able to locally cache such a remote object. If the object is mutable, then the clock is encoded or remotely pointed. If the object is immutable then DM_NULL is sent instead. The entire command returns an Exit item or the associated replica if already present. The *external-options* allows to specify whether the object is copyable, sharable, or uninitialized.

`RECEIVER-REFERENCE` *key* [*unmarshaling command*]

The `RECEIVER-REFERENCE` command returns the object held by the receiving site externalized with that key.

`REMOTE-REFERENCE` *key external-options class clock site* [*unmarshaling command*]

The `REMOTE-REFERENCE` command creates a remote pointer onto an object owned by a remote site. The key is the unique number that names the original object on its birth site. The class may be encoded as a remote pointer itself, the receiving site will fetch it in order to make it locally usable. If the object is mutable, then the clock is encoded or remotely pointed. If the object is immutable then `DM_NULL` is sent instead. The site argument is the site that owns the object. The entire command returns an Exit item or the associated object if already present.

`CACHE` *object exit* [*unmarshaling command*]

This command reads an object and an Exit item, it associates the object to the Exit item and returns the object. This allows, for instance, to pre-send an object in hope it will be useful to the receiver. Something like `cache (limit ...)` *exit*.

Of course, the object must not be already associated to an Exit item and the Exit item must not be already associated to an object.

`SITE` *IPnumber portnumber date site-name-length* [*unmarshaling command*]

The `SITE` command returns a `Site` instance describing the site watching a given port on a given host (specified by its IP number). The `SITE` command is the only command that can create instances of `Site` (this class is uninstantiable by the user). Most often the site is in the not usable state. The site is not contacted and no Connection is opened to it (use `DMeroon_connect` for that). The server is also identified by its birth date (to cope with servers' restart). Since the `Site` class contains an indexed field to hold its host-name, its length is also given.

C.4.5 Technical commands

These commands are internally used by DMeroon.

`DECREMENT` *key object* [*unmarshaling command*]

This command reads a key and decrements the reference counter of the associated Entry item. The Entry item will be reclaimed if the reference counter reaches zero. Eventually the command then reads and returns the object that follows.

`UPDATE-CLOCK` *clock nat4 object* [*unmarshaling command*]

This command is used to mention that the clock appearing as first argument has at least the next *nat4* value. Eventually the command reads and returns the object that follows.

`DEFINE-EMITTER` *site key* [*unmarshaling commands*]

This command is used while bootstrapping a Connection. When a Connection is opened, the requesting Site signals itself with the `DEFINE-EMITTER` command followed by an encoding of the site and followed again with its key. This allows to build on the receiver site, the Site instance and its associated proxy.

Instances of `Site` pass through various peculiar states, see appendix B.

C.5 Object-based protocol

The DMeroon inner protocol specifies the exchange language. When it is possible to exchange objects, it is possible to make DMeroon servers cooperate exchanging structured requests. A request is an instance of the `Request` class. Often, requests are answered by instances of `Answer`, a sub-class of `Request`.

There are several types of requests:

- SCR for `SiteConnectionRequest`. This request is only used to bootstrap a connection. The connecting site sends such a request which is answered by a `SiteConnectionAnswer` (aka SCA).
- OSR for `ObjectSendRequest`. When sending a remote reference towards a site, this request is used. There is no associated answer.
- OFR for `ObjectFetchRequest`. This request is sent when a site wants to refresh a replica. It is answered by an instance of `ObjectFetchAnswer` (aka OFA).
- OMR for `ObjectMutationRequest`. This request is send to the owner site of an object so it can mutate a regular field of this object. It is answered by an `ObjectMutationAnswer` (aka OFA) that returns the former value of the mutated field.
- OIMR for `ObjectIndexedMutationRequest`. This request is send to the owner site of an object so it can mutate an indexed field of this object. It is answered by an `ObjectIndexedMutationAnswer` (aka OIMA) that returns the former value of the mutated field.

<< Section under construction >>

C.6 DMeroon and http

Normally, DMeroon starts an http server. You may invoke it to inspect the site object of DMeroon (if, of course, sufficient time is given to the server to watch for the incoming requests). The default URL is: `http://localhost:56423`. The rest is self-explanatory and clickable as soon as your favorite http client (`xmosaic`, `Netscape`, `IExplorer` etc.) can open this URL.

The site object is the most prominent object of DMeroon. It allows to inquire the exact definitions of DMeroon types, the predefined classes and the published information. The site object contains a "`information`" field containing instances of `Dictionary`, a sort of multi-Association-list containing whatever values exported from this site. Currently there are more than a hundred published values. The published information is publicly available since any user that connects to this site may read the site object and inquire its published information.

<< Section under construction >>

C.6.1 The http protocol

This protocol is triggered by the `GET` keyword followed by an DMEROONSCRIPT request. This request is made of words separated by slashes. It may use the *%dd* convention to encode some characters with their ASCII code. The default URL is:

```
http://localhost:56423/help
```

In fact a whole language exists to write these URLs. This is a small stack-oriented language, the DMEROONSCRIPT language (see chapter 5, that allows to call all the API functions.

C.6.2 Publishing informations

The site object, whose class is `Site`, exports all its content and in particular an immutable field named `information` whose content is a multi-A-list to which you may add your own values. The objects that are reachable from this `information` field is said to be published.

As seen from table C.1, there are initially two dictionaries. The `user` dictionary is empty and reserved for users, the `DMeroon` dictionary is reserved for DMEROON and holds values describing the implementation.

`DMeroon`	*The own* DMEROON *dictionary*
`  site`	*the site object itself*
`  classes`	*the* `Vector` *of predefined classes*
`  constants`	*a subdictionary listing some constants*
`    nil`	
`    true`	
`    ...`	
`  types`	*the* `Vector` *of predefined types*
`  ...`	
`user`	*a dictionary free for users*

Table C.1: Predefined published informations

The `Dictionary` class is defined as:

`Dictionary`		[*Class*]
`rest`	*a mutable* `reference` *to another dictionary or NULL*	
`value`	*a mutable* `reference` *to a* DMEROON *instance*	
`name`	*a repetition of characters*	

Published informations are gathered in nodes from class `Dictionary`. A `Dictionary` is somewhat similar to an Alist node. The `rest` and `value` fields are mutable. The `value` field can of course refer to a `Dictionary` instance justifying the multi-A-list term.

Published objects are known by pathname describing the path to follow through dictionaries to reach these objects. It is not recommended to use names with spaces, dots and slashes, since pathnames are often written as sequences of names separated by slashes, dots or spaces. The easiest way to populate these dictionaries i.e., to publish information is to use the Scheme binding, see 4.

> **Implementation note:** *An http client is not really part of* DMEROON *space since it is connectionless. An html page may have some links towards* DMEROON *objects. These links may be broken if the* DMEROON *server disappears meanwhile.*

Appendix D

Driving DMeroon servers

This chapter describes some applications using the DMeroonScript language to drive DMeroon servers.

D.1 Measuring the progress of a remote process

<< Section under construction >>

Suppose you started a process on a remote machine that produces lines of results that are further processed. Suppose this process to be very long and to require days and days for its completion: you probably have to stop the `rlogin` connection you used to start it and then you can no longer observe its output. A simple solution to measure whether the process makes some progress is to filter the output with a `dmtee` command. This command records the last filtered lines in a DMeroon object. You can then inspect that object with any http client from your office, your home or even from Mars.

Implementation note: *The* `dislocate` *utility, from Don Libes, command seems to be similar.*

The `dmtee` has the following syntax:

`dmtee.prl` *[-n] [host[:port] [pathname]]* [*Command*]

This command creates a connection to the DMeroon server mentioned in the argument *host:port* or, `localhost:56423` by default. This DMeroon should of course be running! Then it creates a mutable vector of size n (by default, 5) keeping the last n lines. This object is published with *pathname* as name or `/user/$USER/$$` by default.

This object can be inquired with the following URL:

`http://`*host*`:`*port*`/`*pathname*

To return to our introductory example, suppose your command was originally `solve | verify >file`, then you just have to transform it into `solve | dmtee | verify >file` to benefit of this new facility.

`dmtee` is a simple Perl script [WS90] using the DMeroonScript language to drive a DMeroon server. It appears in *TOP*`/DMeroon/Commands/dmtee.prl` file.

D.2 Distributed X cut buffer

One day, working on a workstation near Luc Moreau, he asked me for a bibliographical reference, I highlight the text in my favorite Emacs and started to move the selection before realizing that there was no way to paste it on Luc's screen. I then use E-mail. This application allows different users to share X selections. It is written in Tcl and uses the DMeroonScript language.

<< Section under construction >>