DMEROON
Overview of a Distributed Class-based Causally-coherent Data Model [*]

Christian Queinnec[**]
École Polytechnique & INRIA-Rocquencourt

Abstract. DMEROON is a library of C functions that provides a data model above a coherently distributed shared memory. DMEROON allows users to statically or dynamically create new classes, 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, migrating and sharing objects, local and global garbage collections. This document provides an overview of DMEROON.

1 Goals

These days, the net becomes more and more fundamental to nearly all the facets of our professional life, both to use and to study. In the same time, programming languages cease to stay the focal point of our attention. Although important since they are still the mandatory means between machines and developers, their algorithmical side tend to be marginalized in favor of their (tele-)communicational or *interoperability* aspect. This is so because programming languages are generally sequential-minded whereas the net is inherently distributed, more persistent but able to fail.

API (Application Programming Interface) are now of paramount importance, they play the intercessory rôle between developers and some set of functionalities. API start as subroutines libraries but soon evolve to become complex universes involving various kinds of tiny languages, communication protocols, data structures, programming invariants, computational models etc. Compare, for instance `curses` and X!

Nowadays API nearly replace programming languages and propose new models of programmation with higher-level, richer, more complex but more specialized entities. Designing a good API may be compared to the design of a new programming language but is often far more open and difficult since it encompasses far more aspects. For all these reasons, a good API must provide a clear

[*] Revision: 1.14 typeset on March 4, 1997 at 21:23 — PSLS '95, Beaune (France), October 1995.

[**] Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France – Email: `Christian.Queinnec@polytechnique.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS and by the CEC/HCM VIM project.

model of the entities it allows to handle, control or customize. A formal specification or semantics is desirable but probably of extreme difficulty due to the too numerous facets of the API.

Moreover we believe APIs to be potentially more able to synergize applications wanting to cooperate rather than parallel and/or distributed languages that are, years after their inception, still not widely spread.

DMeroon addresses the problem of interoperability defined as the possibility of exchanging and sharing simple or complex data, their identity, relationship and evolution. DMeroon provides a distributed shared memory (DSM) of structured objects. Shared memory is generally recognized as the simplest programming model for distributed programmation. DSM allows developers to program in terms of shared memory rather than in terms of messages, this difference might be compared to the gap that exists between direct style and the low-level Continuation Passing Style as used, for instance, in actor-based languages.

DSM is simpler but only if the provided coherence naturally fits the semantics of the used programming language. DMeroon ensures causal coherency using a lazy invalidation protocol (more generally, laziness is often used as a programming technique inside DMeroon). DMeroon is tailored for wide area networks and thus prefers to limit, for its own implementation, cyclic data, backpointers and so forth.

DMeroon is class-based; objects have classes which themselves are objects (obeying to the ObjVlisp (single inheritance) model and allowing reflection of their structure). An object can be as simple as a float number or as complex as, for example, a bytecoded Emacs package (a vector of bytes accompanied by a number of unrestricted quoted Sexpressions). Objects are distributed on an individual basis, can be remotely read, locally cached but are managed by a single owner that sequentializes mutations.

From a programmer's point of view (see figure 1), DMeroon allows to define classes of objects, to instantiate them, to read or write the fields of these instances, to be connected with other DMeroon sites and to share objects with these remote sites. The new and main characteristics of DMeroon are *(i)* its comfortable data model allowing references (pointers), indexed fields and single inheritance, *(ii)* the causality implying that every participating site observes a coherent data state.

DMeroon has a very dynamic behavior i.e., instances and classes are mainly built at run-time then moved to repositories to acquire more persistence. DMeroon is not encumbered with stub compilers, interface definition language etc. DMeroon relies on reflection for its own self-description. DMeroon is offered as an API and, to be worth of value, tries to satisfy additional constraints such as hardware-, system-, language- and implementation- independence (the latest are the most complex). The language into which DMeroon is incorporated is called the *support language*. DMeroon tries as much as possible not to duplicate the facilities offered by the support language: Memory Management and particularly Garbage Collection (GC) is reused if provided and suitable.

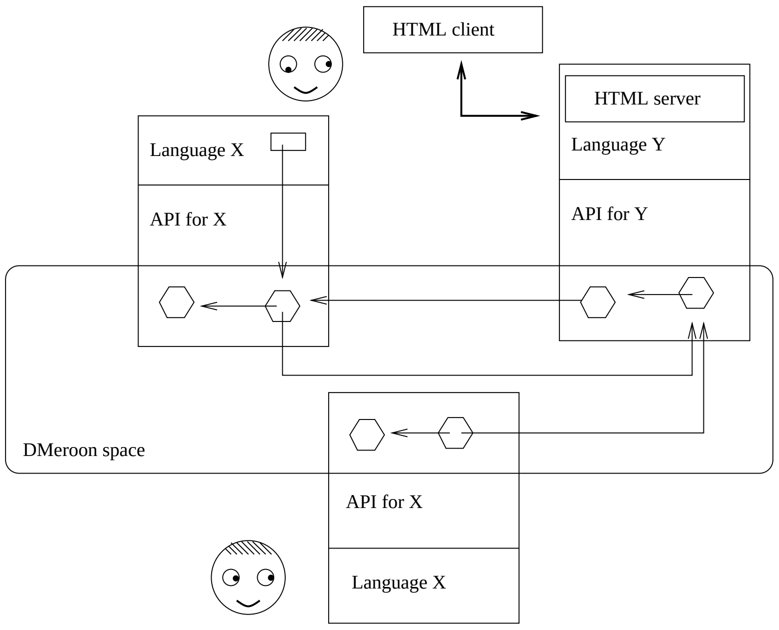

Fig. 1. DMEROON: a distributed shared memory of objects

Albeit DMEROON offers classes, it does not offer generic function nor method invocation nor any control-related functionality. A generic function is an aggregate of methods (functions or procedures), a data type which is not universally understood. DMEROON is only a data model on top of which additional layers can be built up, for instance, to implement a programming language. DMEROON aims to be the smallest multi-language data model library.

This paper describes the current state of DMEROON, its computation model (§2) and its entities (§3). A comparison with related work (§4) and a conclusion finishes the paper.

2 Computing

Any running process incorporating DMEROON is known as a *site*. A site provides a physically independent data space which is part of the whole DMEROON space. The site is reflectively described by an instance of the `Site` class. These site objects reify the perception of space. Any object created on a site is *owned* by that site. The *owner* of an object manages it and, in particular, sequentializes all its evolutions.

A program may follow a reference from a DMEROON object to another DMEROON object without being aware whether this latter object resides on the same site or not. Conversely, any reference to a DMeroon object can be written into a mutable `reference`-typed field of any DMeroon object. Once created, wide and complex graphs of objects can be accessed from any site of DMeroon space.

An instance of the `Site` class has an `information` field holding a mutable multi-level Association-list of DMeroon objects. These objects are said to be *published* and may be accessed from any site. This public information is similar to a kind of `ftp` site (or blackboard or Linda tuple space) where programs cooperate by polling regularly (or according to some other user-defined protocol) the areas where some interesting objects may appear.

All sites initially publish the reference of the world main DMeroon site publishing the source texts of DMeroon, other interesting sites, etc. This computation model enhances the WWW world with objects that are programmatically usable and whose structure may be tailored to users' needs.

Another computation model allows to send (a reference of) a DMeroon object to another site. This allows to asynchronously send/receive objects. It restricts communication to be point-to-point between two sites (or two applications or users standing behind these sites). With a reference, a site may access the fields of the referenced object and consecutively all its offspring. Such an exported object may encode a request such as a mail exchange, a file transfer or a clock synchronization etc. More generally, this computation model provides a message layer where exchanged messages are constrained to be DMeroon objects over a shared memory. But to let DMeroon be aware of these communications allows it to maintain the coherency of the DMeroon space.

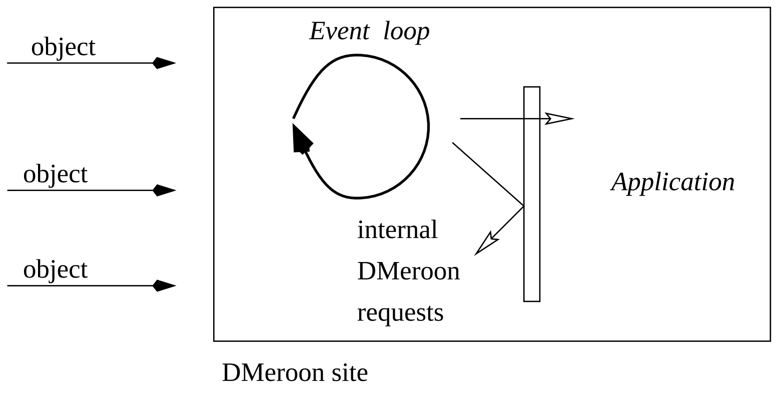

Fig. 2. Event loop on a site

With point-to-point communications, users may, for example, register a function that will be called on any received object (except DMeroon objects that used for internal management, see figure 2). Users may then exchange objects, decode them according to their class or content and perform what they ask for. It is therefore possible to implement all kinds of protocols with the sole operation of sending the reference of an object from one site to another one. DMeroon appears as the ultimate distributed machine that, as any other machine, provides memory management and information transfer.

DMeroon itself uses this mechanism for its own management. When a site sends a request and waits for an answer, the "continuation" of that request is another object containing all the data that are necessary for the resumption of the request. This continuation object is reachable through the request object and serves to identify the answer that matches it. The continuation object may also appear as a place-holder for an expected returned value.

To have messages encoded as regular objects has some additional virtue that may be exploited for fault-tolerance. Copies of sent objects may be kept on their emitting site until reception is acknowledged: the sent object has a reference onto the kept copy, the acknowledgment is simply achieved when the sent object is received and reclaimed, the propagation of that reclamation will ultimately collect the now useless copy.

We believe that DMeroon is appropriate for distributed symbolic computing which is mainly characterized by *(i)* the ability to compute upon complex objects with involved relationship (therefore accompanied by an irregular control) and *(ii)* the ability to offer automatic memory management. DMeroon will probably be used for experimental languages or applications saving them from the burden of managing a distributed shared memory. This is possible since, to be offered as a library with an API, allows DMeroon to be used in multiple contexts without too many constraints.

3 Objects

DMeroon holds its name from Meroon [Que91], an object system for Scheme which introduced a data model powerful enough to represent all Scheme values including vectors and strings in an uniform framework. A DMeroon object is a contiguous piece of memory containing the values of the fields of its class, see figure 3. Fields may be *regular* or *indexed* without inheritance restriction. 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 allocation time. Since the kernel of DMeroon is written in C, DMeroon objects are physically represented with the conventions (size, alignment) of the used C compiler.

3.1 Fields

All fields have a precise type. There are signed or unsigned fix numbers with specified range as well as float numbers and characters. There is also the `reference` type that may hold a pointer to any DMeroon object whether local or not.

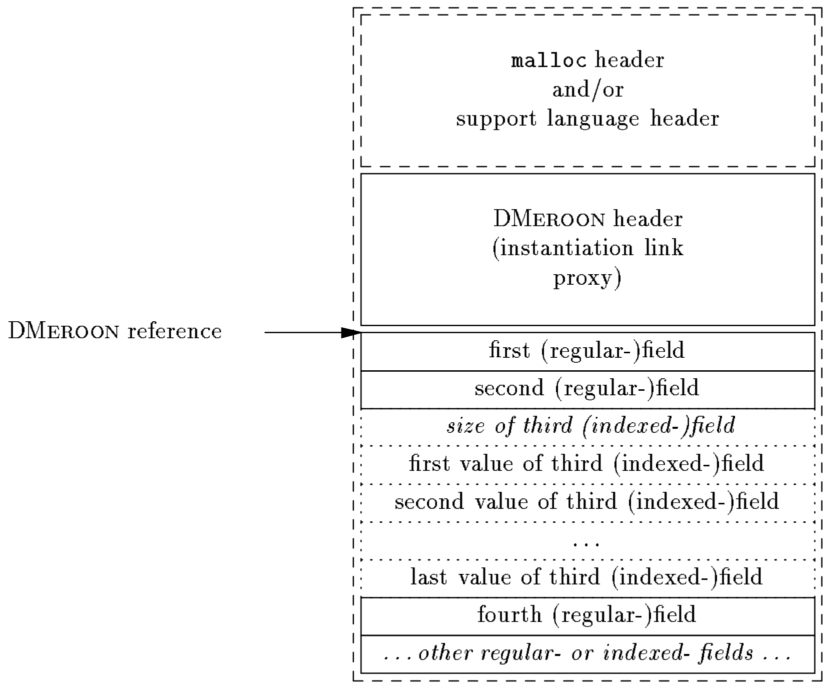
malloc header
and/or
support language header
DMeroon header
(instantiation link
proxy)
DMeroon reference
first (regular-)field
second (regular-)field
size of third (indexed-)field
first value of third (indexed-)field
second value of third (indexed-)field
...
last value of third (indexed-)field
fourth (regular-)field
... other regular- or indexed- fields ...

Fig. 3. DMeroon instance example

Fields may have properties that are specified at class creation time. These properties are inherited by subclasses without change. When a field is *mutable*, the user is allowed to modify it. When a field is *volatile*, its content cannot be cached and must be fetched any time it is read even if remote. When a field is *local*, its content is only meaningful for the current site and therefore is never transmitted. These properties may be arbitrarily combined. The volatile property is mainly used for an immutable object whose content is modified by the implementation: the `time` field of a `Clock`, for instance. The local property generally qualifies secret (non exportable) fields of the implementation.

Indexed fields are essential: they allow to embed sequences of values contiguously within objects. Sequences cannot be separated from their embedding objects, they avoid intermediate references and are not transmitted independently. DMeroon data model is particularly comfortable since, in addition to regular records (dotted pairs for example), it allows to express vectors (sequences of references) or strings (sequences of characters) as regular classes and not as special primitive non-subclassable classes. The Icslas language [Que93] is entirely supported by the DMeroon data model and therefore does not hide the structure of its basic values.

DMeroon's API offers some accessors. It is possible to determine the length of an indexed field, to read a regular or indexed field (the latter must mention a correct index) and to write a mutable regular or indexed field. Mutation is atomic; mutators come with two flavors: one returns the previous content of the field (the initial value of a field is unspecified), the other just perform the mutation and returns no value.

Fields are reified into `Field` instances describing the nature of the field (regular or indexed), the type of the field (and of its index if any), the class that introduced that field and some precomputed information on how to access that field. Field descriptors may be accessed through objects' class.

Since access is performed through function invocation, it's price is high but this mechanism ensures that all read/write accesses are perceived by DMeroon to ensure the coherency of the DSM.

3.2 Structure

DMeroon objects have a specific DMeroon header currently limited to two references. The first one implements the instantiation link and is a pointer onto the class of the object. The other one contains the data necessary for remote access or, more precisely a reference to an `Entry` or `Exit` object as shown on figure 4. These `Entry` or `Exit` objects are not visible from the user, they are only used by DMeroon itself.

DMeroon is class-based, it only offers single inheritance since it is a well-understood model with a clear and unambiguous semantics; multiple inheritance would have required to choose among the possible choices for slot inheritance, moreover full adherence to C++ was not felt to be an unavoidable requirement.

DMeroon's API allows to obtain the class of any object. It is also possible to check whether a class is a subclass of another one. Since classes are themselves objects, they may be inquired as any other objects. A class has a name for debug purpose (there is no predefined way to convert a name into a class), a sequence of super-classes and a sequence of field descriptors as well as some precomputed information describing its main characteristics (mutable, containing references, etc.)

A peculiarity of DMeroon is that classes do not have a `subclasses` field holding their subclasses. First, it avoids a mutable field, second, it removes a cycle between classes and their superclasses, and third, it offers *ubiquitous* objects such as `Object` the class at the root of the inheritance tree. Ubiquitous objects are immutable and indistinguishable objects shared by all DMeroon sites.

Objects are central to DMeroon even without methods: they have a reflected structure which is independent (and more persistent) of any further treatments. As more and more objects are published, newer and newer ways of processing them will occur unimpeded by any higher-level philosophy of use. This is why DMeroon adopts low-level goals.

3.3 Allocation

Allocation faces two problems. Since DMeroon only exists via a support language, DMeroon objects are embedded inside values of the support language and therefore must be allocated (on the current site) by (or compatibly with) the specialized allocation routines of this support language. This also imposes useless DMEROON objects to be reclaimed by the GC of the support language if any (otherwise, DMEROON brings its own GC which is currently Boehm's GC [BW88]). Finally to achieve non-local GC over part of the DMEROON space [LQP92] requires to be aware of the `Exit` instances that are reclaimed by the GC of the support language.

The second problem concerns the initialization of DMEROON objects. Most of the initialization protocols of other Object Systems use mutation to achieve initialization. The cost of coherence maintainance is so high that mutability must be kept to a minimum and clearly, initialization is not a mutation. Moreover since DMEROON only deals with data, the initialization protocol cannot be based on procedures which are not DMEROON objects. Finally the solution must also solve the co-instantiation problem [Que93] i.e., the allocation of mutually referencing objects without mutable fields.

The solution is very crude. A raw allocator exists that takes the lengths of the indexed fields and returns an object with unspecified fields content (but specified structure). Such an object is considered to be *under initialization*. While in that state, it cannot be exported but all its fields can be written. This state is ended when the user notifies DMEROON that the object is initialized. For sake of reflectivity, a predicate allows to know whether an object is under initialization.

This initialization protocol is sufficient to build the usual class-specific allocation procedures. Another allocation procedure of DMEROON is `create-class` which for convenience returns fully initialized classes. `create-class` had been individualized since class creation is an example of co-instantiation where the `Class` object refers to its proper `Fields` that themselves refer to the `Class` that introduced them. Some information is also precomputed at class creation time and recorded into the fresh instances.

3.4 Coherency

When a mutable object i.e., an object containing at least one mutable field, is *exported* i.e., is referenced from remote sites, it is always *monitored* by a *neighbor* clock i.e., a clock owned by the same site, see figure 4. A clock counts the number of modifications that occurred to the (possibly multiple) objects it monitors. To speed up read accesses, remote objects are cached and clocks are used to determine if the cache is valid. When a communication occurs between two sites, the receiving site takes notice of the values of the clocks known by the emitting site and updates its own view of these clocks: this is a lazy propagation of clocks values. When a cache is filled, it records the value of the clock monitoring the remotely read object. Therefore a cache is invalid if its monitoring clock is known to be greater. This implements a lazy invalidation protocol and ensures causality i.e., when a site receives information from another site, it cannot ignore the mutations this other site was aware of. See [Que94b, Que94a] for more details.

Causality seems to be the weakest coherency protocol that still keeps a decent semantics for a distributed memory viewed from a language, that is, offered by

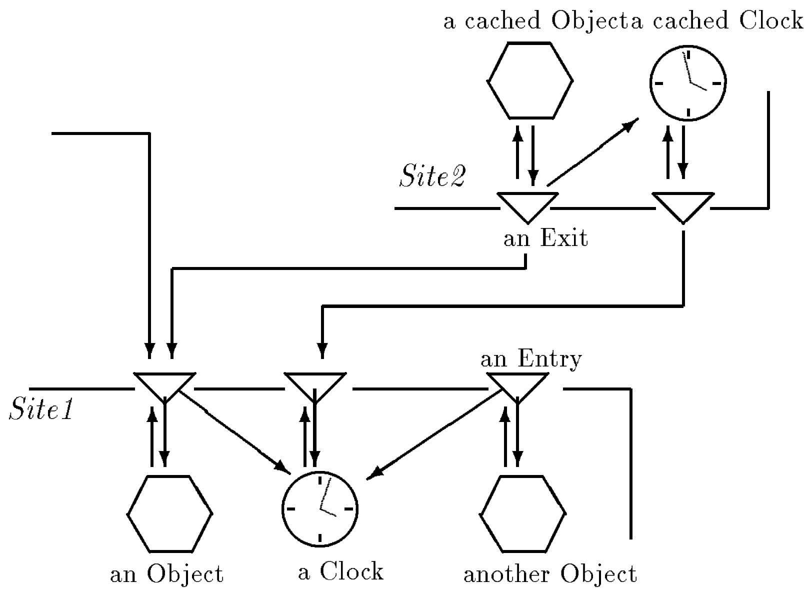

Fig. 4. DMEROON remote reference and clocks

an API.

3.5 Sharing

Of course the real interest of DMEROON it the ability to let objects be shared. The basic mechanism is the following, see figure 5. When a site (Site2 on the figure) has a reference onto an object of a remote site and the user asks for its content: first, the site checks whether the class of that object is locally known. If this is not the case, then the class is asked for (and of course recursively its proper class, metaclass, fields, class of fields etc.) Once the class is present then the stream of bytes corresponding to the marshaling of the original object can be decoded and the content of the object can be locally cached and read. If the object is mutable then the current time of its monitoring clock also accompany its content.

DMEROON's API makes it is possible to send the reference of an object to a site. The application of the receiving site may ask (or wait) for an object to arrive. DMEROON ensures `eq`-*ness* of objects i.e., if an object is exported and (more or less directly) sent back to its owning site, the received object is `eq` to the original object. This property confers objects an identity that may be used (this is what is required for predefined classes to be ubiquitous). This property avoids overflowing a site with multiple copies of a same object and allows to trace back objects to their originating site for failure recovery.

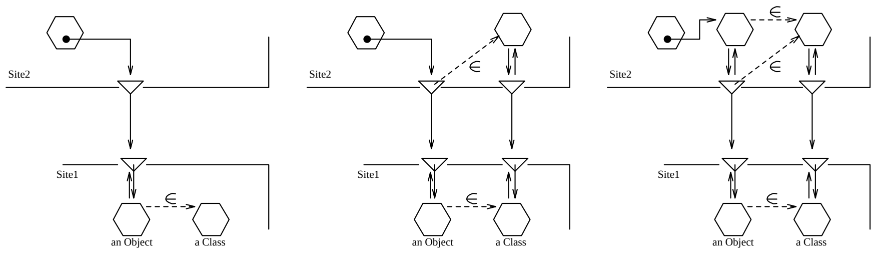

Fig. 5. Exportation (class exported before object)

3.6 Clocks

DMeroon's API offers to specify explicitly the clock that monitors an object. On all sites there exists a general clock which, by default, monitors all exported objects not already monitored by a clock. Because a clock monitors numerous objects, and since the lazy invalidation protocol [Que94b, Que94a] when incrementing a clock, *ipso facto* invalidates all its cached objects, one may create clocks and explicitly specify the objects they monitor. To have numerous clocks allows to invalidate less objects when ticking but increases the transmission overhead since more clocks are (lazily) spread over DMeroon space.

A clock can only monitors initialized mutable *neighbor* objects i.e., objects owned by the site owning the clock. It is possible to change the clock that monitors an object. If the new clock is a neighbor, it is set to monitor the object and the former clock is ticked to force caching sites to notice this change. If the new clock is not a neighbor then the object has to be implicitly migrated to the site owning this clock.

3.7 Migration

DMeroon's API also offers the opportunity to know the owning site of an object. *Migration* is also offered; migration allows to change the owner of an object. Details on this protocol is omitted (it is inspired from [Piq91] with additional care of clocks). Migration is useful for various purposes: *(i)* to move objects closer from computations or, *(ii)* when shutting down a site, to preserve exported objects (and their offspring) onto another (repository) site.

A class may be created with the *immotile* property to prevent its instances from being migrated. This is for instance the case for all internal DMeroon objects such as Tcp connections, `Exit` or `Entry` entities etc. This improves security with faithless DMeroon implementations: an immotile immutable object owned by a faithful DMeroon implementation cannot be perverted.

3.8 Extending space

The last operation provided by DMeroon allows to explicitly add another site to the whole DMeroon space containing the current site. The new site is specified by an IP address and a port number. When this new site is made reachable, it is reified into an instance of the `Site` class that can be used to identify this site when transmitting (references of) objects.

4 Comparison

DMeroon provides a distributed shared memory above which computations may be done. Although DMeroon is not yet publicly available, we may compare it with Tcl-DP, ILU and Network Objects. There are many other distributed shared memory systems distributing pages instead of objects that we do not consider. Similarly we also ignore the many object-based languages with distribution capabilities.

Tcl-DP [RS94] is an extension of Tcl/Tk standing for Tcl Distributed Programming. Tcl-DP adds TCP and IP connection management, remote procedure call (RPC), and distributed object protocols to Tcl/Tk. DMeroon improves on Tcl-DP on several points: *(i)* objects are anonymous, *(ii)* they may contain indexed sequences, *(iii)* they do not need to be explicitly distributed, *(iv)* access to local or distributed objects is the same, *(v)* the graph of distribution is not restricted to be acyclic, *(vi)* deallocation is automatic. Conversely, Tcl-DP allows objects to be extended with new slots, to specify default values for slots, to associate before- or after- daemons to slot access. Nevertheless Tcl as a binding language for DMeroon is envisioned and may be used to provide generic functions and additional features such as these brought by Tcl-DP.

ILU [JSS95] is a library of programs providing a mechanism of remote method invocation very close to CORBA. ILU is very complete and has been ported to many machines. An object in ILU is a set of methods described by an interface written in the ILU Interface Specification Language from which are automatically generated stubs for Common Lisp, C, C++, Modula 3 and Python. DMeroon differs from ILU on several points: *(i)* DMeroon is dynamic and does not require an interface language nor stub generation, *(ii)* DMeroon offers raw access to objects, `eq`-ness of objects and some reflective capabilities. *(iii)* DMeroon provides a shared memory with causal coherence. Of course, these advantages are also its disadvantages since DMeroon is less efficient and does not offer remote method invocation.

Network objects are described in [BNOW94] and serve as a basis for Obliq [Car95]. They implement objects paired with methods in the framework of Modula-3 and provide a strongly typed remote method invocation facility. Modula-3 manages its memory including network objects with a GC. Differences with DMeroon are: network objects do not manage coherency, do not provide non local GC, are only supported by Modula-3. Nevertheless, the goals and some choices of network objects are very close to ours.

5 Conclusion

This paper is the first overview of DMeroon, a distributed shared memory with causal coherence. Its API is described and some design choices are commented. To sum it up: simplicity of the API was a major goal, reflectivity of the architecture makes it rather open, its low-level approach data-focused should normally make it a good vehicle for interoperability.

Last but most unfortunately, DMeroon is still under progress, may evolve (as new support languages will be considered such as Tcl, Python, Emacs, C++ ...), will unquestionably discover new problems (at least: persistence, fault-tolerance and communication-efficiency) and is not yet available! Fresh information may be obtained from URL:

`ftp://ftp.inria.fr/INRIA/Projects/icsla/WWW/DMeroon.html`

Bibliography

[BNOW94] A D Birrell, G Nelson, S Owicki, and E Wobber. Network objects. Technical Report 115, DEC – SRC, 1994.

[BW88] Hans J Boehm and M Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9), September 1988.

[Car95] Luca Cardelli. A language with distributed scope. In *POPL '95 – Twenty-second Annual ACM symposium on Principles of Programming Languages*, pages 286–297, San Francisco, California, January 1995.

[JSS95] Bill Janssen, Denis Severson, and Mike Spreitzer. *Inter-Language Unification – ILU 1.8 Reference Manual.* ftp://ftp.parc.xerox.com/pub/ilu, 1.8 edition, 1995.

[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.

[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.

[RS94] Lawrence A Rowe and Brian C Smith. *Tcl Distributed Programming (Tcl-DP)*. ftp://toe.berkeley.edu/pub/multimedia/Tcp-DP, 3.1 edition, February 1994.