Sérialisation–désérialisation en DMEROON

Christian Queinnec

LIP6 & INRIA-Rocquencourt

RÉSUMÉ — La transmission de valeurs entre mémoires séparées passe par une sérialisation qui les transforme en un train d'octets puis par une désérialisation qui retransforme ces octets en valeurs équivalentes. Ces deux procédés sont généralement conçus, écrits et maintenus de concert : toute amélioration nécessite de les faire évoluer tous deux. Nous proposons ici de considérer la sérialisation comme une compilation transformant une valeur en un programme délivré, pour évaluation, à un interprète de désérialisation. Le protocole est alors défini comme le langage d'échange. La richesse de ce dernier fonde la souplesse de ce premier puisqu'une large gamme de compilations est possible selon les particularités des valeurs à transmettre. Nous relatons cette expérience dans le cadre du système de mémoire partagée répartie DMEROON.
MOT-CLÉS : *sérialisation, transmission de valeurs.*

ABSTRACT — Transmitting values between distributed address spaces requires marshaling these values into a stream of bytes; this stream is then unmarshaled into equivalent values. Quite often, these two processes are simultaneously conceived, written (or generated) and maintained. Any improvement on one side has to be reflected on the other side. This paper proposes to consider marshaling as a compilation of a value into a program (a bytecode program) transmitted for evaluation to a unmarshaling bytecode interpreter. The marshaling protocol is therefore defined as the language as defined by its compiler and run-time interpreter. When the language is rich enough, the protocol is very versatile since many possible compilations are possible depending on the peculiarities of the data to transmit. The Distributed Shared Memory DMEROON system uses this technique.
KEY WORDS : *serialization, marshaling.*

Afin de communiquer, les systèmes répartis s'échangent des valeurs structurées. Pour les valeurs terminales comme les entiers, les flottants, les caractères ou pour les répétitions bornées de telles valeurs, il n'y a guère de problèmes. XDR[1], par exemple, réalise cela depuis de nombreuses années, notamment au sein de NFS[2] et du mécanisme des RPC[3]. La norme ISO[4] intitulée ASN.1[5] définit de façon similaire un langage de description de données et son encodage. Corba™[6][7] fournit de plus un IDL[8] qui explicite les signatures des méthodes pouvant être invoquées sur des objets distants. Enfin, Java™[9] surenchérit encore en introduisant

[1] *eXternal Data Representation*
[2] *Network File System*
[3] *Remote Procedure Call*
[4] *International Organization for Standardization*
[5] *Abstract Syntax Notation One*
[6] *Common Object Request Broker Architecture*, marque déposée par OMG®.
[7] Object Management Group
[8] *Interface Definition Language*
[9] marque déposée par *Sun Microsystems, Inc.*

des possibilités de GC[10] (glanage de cellules) avec le protocole RMI[11]. L'abondance voulue de sigles dans ce paragraphe montre tout l'intérêt industriel porté à ce domaine.

Le plus simple des procédés de sérialisation/désérialisation est probablement celui, traditionnellement utilisé dans le monde de Lisp, où les échanges (ou archivages) de données sont effectués au moyen des fonctions génériques `print` et `read`. L'émetteur imprime les valeurs à transmettre dans un flux de caractères lu par le récepteur. La plupart des valeurs, autres que fonctionnelles, disposent en effet d'une syntaxe permettant de les représenter. Cette syntaxe peut en outre exprimer les partages ou cycles de données [Steele, Jr. 90, p. 537]. Un flux de caractères est lisible, portable mais volumineux et coûteux à lire ou imprimer [Lecouffe 79].

La parade logique fut celle qui s'est imposée avec XDR [xdr] qui encode les valeurs en binaire assurant ainsi compacité et vitesse. Seuls sont encodés les types de base du langage C, toutefois, la mise en œuvre habituelle passe par un générateur de sérialiseurs/désérialiseurs (`rpcgen` par exemple) qui convertit des descriptions de données (fichiers `.h` ou `.idl`) en code C spécialisé dont la taille est proportionnelle au nombre de types de données échangés. Afin de réduire la taille de ce code et, surtout, de pouvoir échanger de nouvelles structures de données créées dynamiquement et postérieurement à l'écriture du sérialiseur/désérialiseur, il est nécessaire de recourir à une interprétation (éventuellement compilante [Hof 97]) des descripteurs de données. Cette technique conduit à des interprètes compacts et n'est pas inefficace [Bartoli 97].

Malheureusement cette approche n'assure pas naturellement le respect de la topologie des données de l'émetteur. Ces données sont re-créées dans l'espace mémoire du récepteur et les partages ou cycles originellement présents ont disparu sauf encodage manuel particulier. Cette situation est particulièrement gênante pour les langages de haut niveau ayant quelques soucis sémantiques concernant la notion d'égalité, qui doivent alors assurer que la transmission de deux objets égaux n'invalide pas leur égalité.

Il nous semble que l'arrivée de langages prenant plus explicitement en compte les problèmes liés à la répartition devrait *(i)* rendre généralement invisible la répartition des données, *(ii)* procurer une gestion automatique (avec GC donc) de la mémoire répartie. Le premier but impose notamment de procurer une conception saine de l'égalité, le second implique en outre une certaine cohérence de l'espace mémoire ainsi géré (ce dernier aspect ne sera pas détaillé car en dehors du thème de cet article).

Nous nommerons dorénavant objets, les (aggrégats (souvent contigus) de) valeurs manipulées de façon indivisible par les langages. La classe d'un objet comporte toutes les informations décrivant la structuration du dit objet. L'égalité entre objets que nous adopterons est classique [Baker 93] : deux objets immuables sont égaux si structurellement égaux ; un objet modifiable, d'où qu'il soit vu, n'est égal qu'à lui-même.

Le respect de la topologie des objets à sérialiser en fonction de l'égalité ci-dessus rappelée, impose d'identifier les objets transmis de façon à ne pas les retransmettre (pour gagner en vitesse et en taille de transmission et pour ne pas non plus saturer un site récepteur de multiples copies d'un même objet) tout en assurant les partages originellement présents. À cette fin, on associe usuellement des tables de hachage et/ou des caches au sérialiseur/désérialiseur.

Lors de la réalisation du sérialiseur/désérialiseur de DMeroon [Queinnec 95], le système de mémoire répartie servant de fondation à un langage réparti [Queinnec 93] en cours d'implantation, nous avons remarqué que le sérialiseur était modifié près de deux fois plus souvent que le désérialiseur. Ce dernier a été conçu comme un interprète d'octets qui est doté d'un ensemble d'opérations permettant de construire des objets locaux, de gérer des tables de hachage, des caches, des piles etc. Une fois ces opérations implantées, la sérialisation n'est plus qu'un processus de compilation d'un objet en un programme (un train d'octets) qui sera exécuté par le désérialiseur. La plupart des optimisations aujourd'hui imaginées ont pu être codées, essayées, mesurées sans changement du désérialiseur. Cette technique n'est en rien révolutionnaire mais voir ce processus comme une compilation suivie d'une exécution permet, à interprète constant, d'améliorer le protocole existant sur le plan général ou bien de le spécialiser pour certains types de données : toutes

[10] *Garbage Collection*

[11] *Remote Method Invocation*

qualités d'évolution de bon aloi.

La suite de cet article présentera DMeroon, ses buts et ses contraintes ainsi que son modèle mémoire. Nous détaillerons alors l'implantation des pointeurs distants ainsi que le processus de sérialisation/désérialisation. Nous terminerons par divers points annexes et notamment le délicat problème de l'amorçage de ce processus.

1. Le modèle DMeroon

Selon la taxonomie de [Protić 95], DMeroon est une bibliothèque de fonctions procurant une mémoire répartie au-dessus d'Internet pour assurer un partage cohérent d'objets entre de multiples lecteurs et, à tout moment, au plus un écrivain. La gestion est répartie et la cohérence est de type causale.

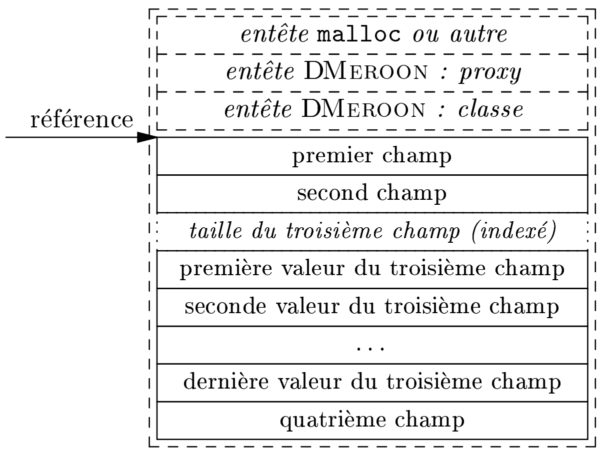

Figure 1. *exemple d'objet de* DMeroon *(Ech : 1/2)*

Un objet est une suite contiguë de champs dont les types ressemblent à ceux du langage C sauf en ce qui concerne leur intervalle de variation qui est dûment spécifié. On y trouve les entiers signés ou non signés, en ordre réseau ou normal sur 1, 2, 4 ou 8 octets, les flottants sur 4 ou 8 octets, les caractères, etc. On y trouve aussi les références à des objets DMeroon et les pointeurs sur des données (`void*`) ou fonctions C. Ces deux derniers types permettent de maintenir des liens entre objets de DMeroon et données ou fonctions d'un utilisateur. Bien que ces valeurs n'aient de sens que localement au site qui les héberge, elles sont néanmoins sérialisables afin d'être utiles dans le cas de communication avec des sites homogènes c'est-à-dire ayant même matériel, même système d'exploitation et même exécutable. Aucune garantie d'interopérabilité n'est cependant assurée pour ces valeurs.

Les objets de DMeroon permettent de regrouper des valeurs qui, ainsi, ne peuvent être communiquées qu'ensemble. Un objet est constitué d'une suite de champs, ces champs peuvent éventuellement être indexés c'est-à-dire dotés d'une taille dynamiquement définie à l'allocation de l'objet (ainsi une chaîne est-elle une répétition de caractères et non une classe prédéfinie). Les champs indexés sont actuellement implantés comme une répétition de valeurs préfixée par leur nombre, voir figure 1[12].

Tout objet de DMeroon possède une classe qui décrit la structure de cet objet. La classe permet de réaliser les calculs d'accès aux champs (éventuellement indexés) de l'objet tout en prenant en compte les problèmes de taille, d'alignement et de droit d'accès (champ modifiable, volatile, local). Les classes sont

[12]Les figures possèdent une indication d'échelle mesurant le degré de détail révélé : une figure à l'échelle 1/2 contient plus de détails qu'une figure à l'échelle 1/10.

elles-mêmes des objets de DMEROON suivant le modèle ObjVlisp [Briot 87]. Les classes sont anonymes et il n'existe aucun moyen prédéfini de convertir un nom en une classe. Inversement, les classes ont un champ indexé de caractères qui permet de leur associer un nom imprimable à des fins de mise au point.

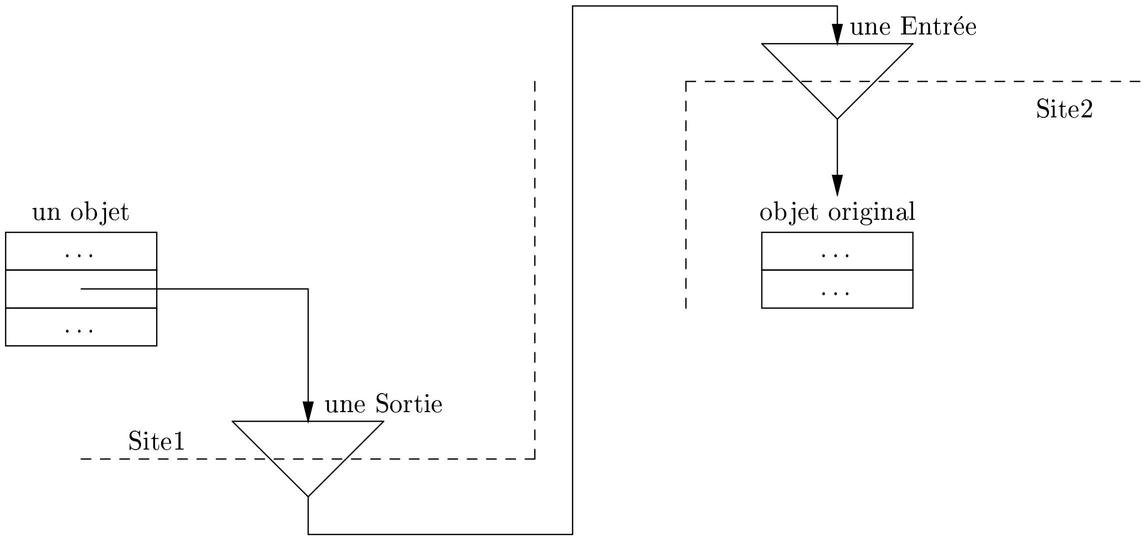

Figure 2. *Pointeur distant (Ech :* 1/10*)*

DMEROON apparaît sous la forme d'une bibliothèque, écrite en C, résolument multilingue (il existe actuellement une interface de programmation pour C et pour quelques implantations de Scheme dont Bigloo et OScheme ; de nouvelles interfaces sont à l'étude pour Caml, Emacs, Java et Tcl). Un programme peut créer dynamiquement de nouvelles classes, les instancier, lire ou écrire les objets, introspecter DMEROON lui-même qui n'utilise que des objets DMEROON pour ses besoins propres, enfin, envoyer des objets à l'intention d'autres programmes, eux aussi liés à DMEROON et s'exécutant sur d'autres sites. Ce côté dynamique est une caractéristique essentielle de DMEROON.

Après allocation et initialisation, un objet peut être déclaré copiable ou partageable. Lorsqu'un objet copiable est envoyé vers un site distant, il est en fait cloné et la copie ainsi lointainement créée ne conserve plus aucun lien avec son original. En revanche, lorsqu'un objet partageable est envoyé à l'intention d'un autre site, ce site reçoit une référence sur cet objet distant, voir figure 2. Le site distant peut alors lire l'objet et stocker localement une réplique de celui-ci qui permet d'accélérer les futures lectures, voir figure 3.

Un protocole de respect de cohérence est alors mis en œuvre [Queinnec 94]. Pour assurer cette cohérence, une réplique conserve des informations permettant de retrouver l'objet original, ces informations servent non seulement au GC [Piquer 91] mais aussi à assurer que la comparaison de répliques est équivalente à la comparaison de leurs originaux associés assurant ainsi une implantation correcte du prédicat d'égalité.

DMEROON a bien d'autres qualités non liées au présent article : nous renvoyons les lecteurs intéressés à la page suivante `ftp://ftp.inria.fr/INRIA/Projects/icsla/WWW/DMeroon.html`.

2. Implantation de DMEROON

Cette section détaille les représentations des pointeurs distants ainsi que les mécanismes qui leurs sont associés. Tout objet a une classe qui décrit sa représentation. Lorsqu'un utilisateur crée un objet grâce aux fonctions de l'interface programmatique, cet objet est local au site dans lequel s'exécute le processus courant. L'utilisateur manipule l'objet via un pointeur qui, comme en C, désigne le premier octet significatif

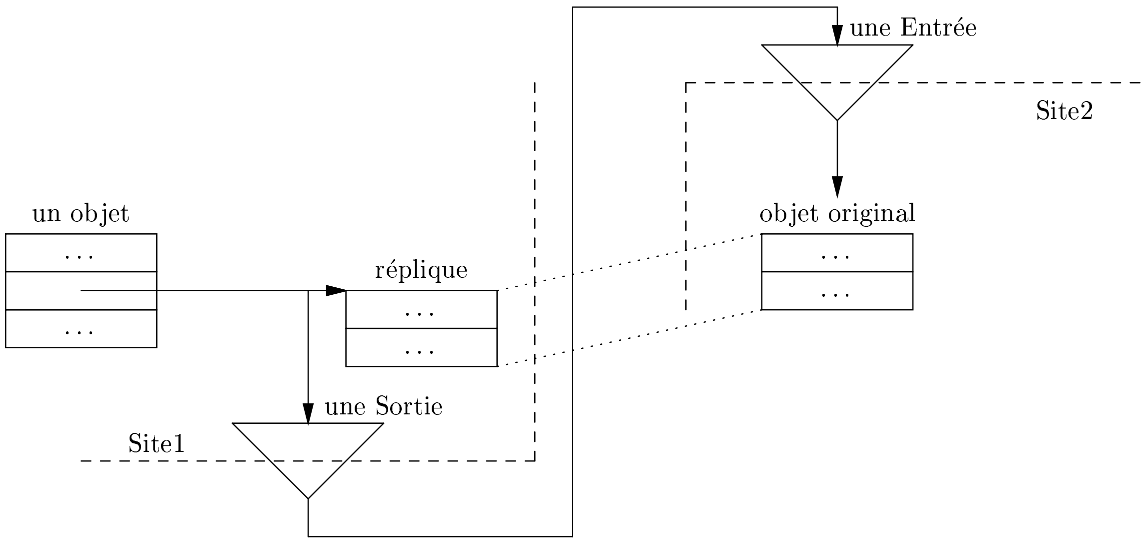

Figure 3. *Pointeur distant et réplique locale (Ech :* 1/10*)*

du premier champ. Deux pointeurs, associés à l'objet, sont situés juste avant cet octet, voir figure 1, et référencent et sa classe et son proxy. Le proxy contient les informations nécessaires à la gestion des pointeurs distants.

Lorsqu'un objet est local, il n'a pas de proxy. Le fardeau dû à DMeroon est donc limité aux deux pointeurs additionnels *proxy* et *classe* que l'on voit en figure 1.

Lorsqu'un objet partageable est externalisé c'est-à-dire qu'une référence sur lui est transmise à un site distant, DMeroon lui associe comme proxy sur le site émetteur, une entrée (un objet de la classe *Entry*). Cette entrée contient (à côté d'informations comme un compteur de référence pour le GC ou un pointeur sur une horloge pour la cohérence (nous ne parlerons dorénavant plus d'horloge, cf. [Queinnec 94])) un numéro identifiant l'objet et un pointeur inverse sur l'objet même, voir figure 4. De plus, chaque fois qu'un objet est externalisé, sa classe l'est également. Toutes les entrées locales à un site sont accessibles par une table de hachage. Une entrée n'externalise qu'un unique objet, un objet n'a au plus qu'un unique proxy.

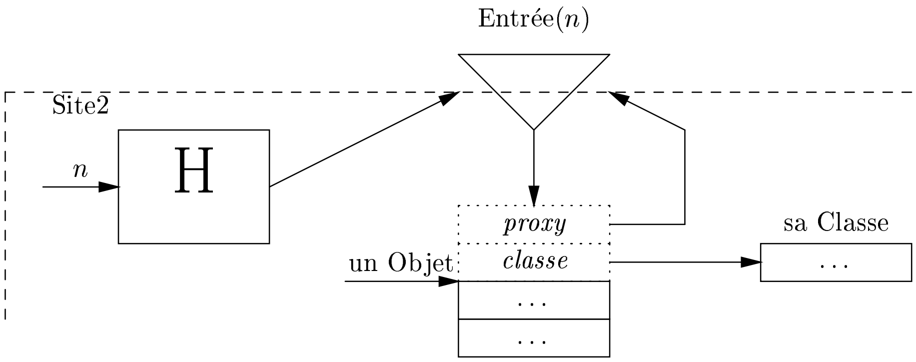

Figure 4. *Un objet référencé lointainement (Ech :* 1/5*)*

Lorsqu'un champ de type référence sur un objet de DMeroon est lu et qu'il contient un pointeur sur une sortie (de la classe *Exit*, sous-classe de *Entry*), une réplique de l'objet original et distant qu'il référence est demandée. Une sortie contient donc toutes les informations nécessaires pour l'obtention de cette réplique

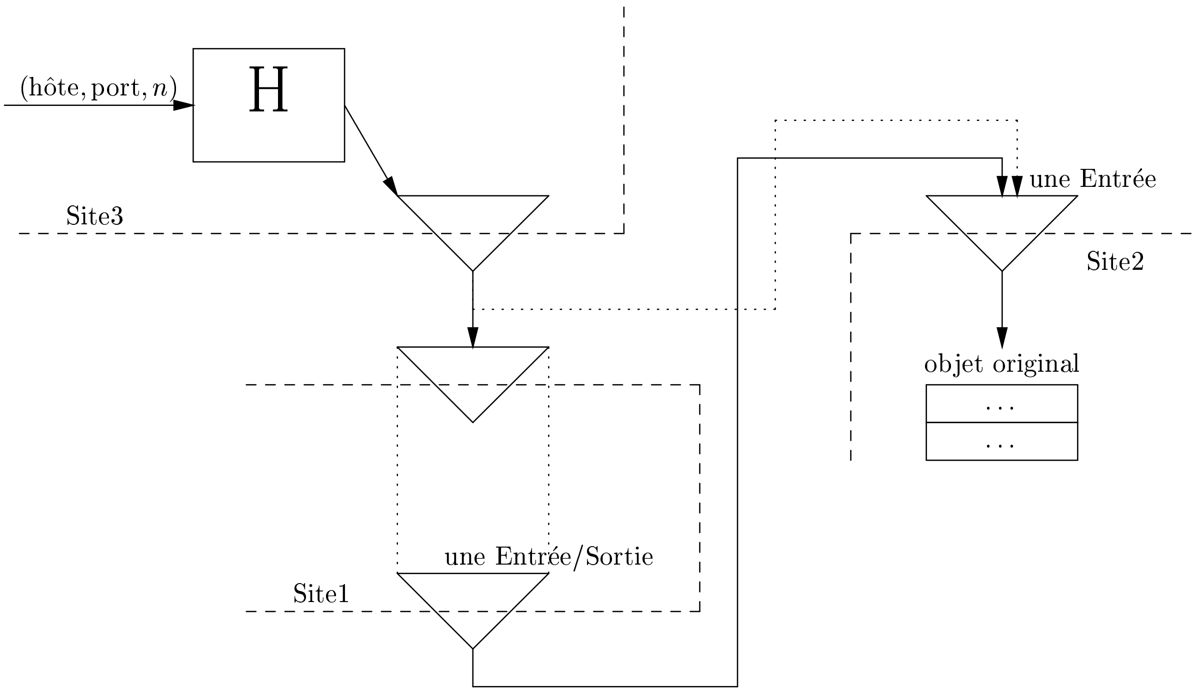

Figure 5. *Pointeur très distant (Ech :* 1/5*)*

et, en particulier, le numéro IP du site distant, le numéro de port Tcp utilisé et le numéro de l'entrée correspondante (une partie de ces informations est en fait factorisée dans des objets de la classe *Site*, voir figure 7). Pour des raisons de GC pour lequel on conserve l'arbre de diffusion des objets, les informations précédentes permettent à la fois de désigner l'objet sur son site original et la réplique par laquelle on l'a obtenue, voir figure 5.

Une sortie a également un pointeur sur la classe de l'objet distant, pointeur dont on verra l'usage plus loin. Un pointeur sur une sortie pourvu d'une réplique locale peut être court-circuité pour pointer directement sur la réplique locale, comme indiqué entre les figures 2 et 3.

Sur un site donné, un objet distant ne peut être référencé que par au plus une sortie : une table de hachage associe une sortie à chaque triplet *(hôte, port, numéro)* la référençant. Il est possible, comme le montre la figure 5, qu'une sortie soit retransmise sans même qu'une réplique locale soit présente : c'est pour cela qu'une sortie est potentiellement aussi une entrée (en d'autres termes, la classe *Exit* hérite de la classe *Entry*). Maintenir la structure de diffusion de l'objet est utile pour le GC par contre les lectures ou écritures demandées par Site3 n'ont pas besoin de transiter par Site2 pour être traitées par Site1.

La migration d'un objet, lorsque son original change de site d'hébergement, s'effectue similairement à [Piquer 91] en laissant un pointeur de renvoi et en raccourcissant les chaînes d'indirection [Shapiro 92, Moreau 97]. Par ailleurs, la table de hachage des sorties ne doit pas être une racine pour le GC local puisque l'on doit détecter les sorties qui sont inutilisées afin de libérer les objets distants. Ceci est réalisé, pour le GC de Boehm [Boehm 88], en déguisant les pointeurs de cette table afin qu'ils ne ressemblent plus à des pointeurs.

3. Transfert d'objets

Cette section présente les mécanismes de transfert d'objets. La classe d'un objet comprend la description de ses champs. On utilise donc la classe pour sérialiser ou désérialiser les objets. Corollairement, on ne peut désérialiser un objet que si le récepteur dispose de sa classe. Lorsque la classe est prédéfinie c'est-à-dire appartient à la cinquantaine de classes présentes dans toutes les implantations de DMeroon, il n'y a pas de problème. En revanche, pour une classe créée dynamiquement sur un site, il faut prendre quelques précautions pour échanger des objets de cette classe.

Pour transmettre un objet, on transmet au minimum de quoi bâtir un pointeur distant sur cet objet et l'on force le récepteur, s'il désire le contenu de cet objet (référencé donc par une sortie sur le site récepteur), à le demander de la manière suivante :

1. il vérifie que la classe est utilisable c'est-à-dire,
   (a) localement connue (c'est pour cette raison qu'une sortie a un champ pour contenir la classe de l'objet distant),
   (b) telle que tous ses descripteurs de champs soient également connus (ce sont des objets de la classe *Field*).
2. il demande enfin le contenu de l'objet puisqu'il est sûr alors de pouvoir le désérialiser.

Comme les classes sont des objets normaux de DMeroon et que, pratiquement, toutes les classes de l'implantation sont sous-classables, la classe peut elle-même posséder une méta-classe non standard que le récepteur demandera au préalable et de la même manière que précédemment expliqué: classes, métaclasses, champs etc. sont des objets DMeroon normaux. Ce schéma de diffusion paresseuse des classes permet à tout site d'en créer dynamiquement de nouvelles et de transmettre sans restriction des instances de ces classes. Le partage d'objets entre plusieurs utilisateurs est ainsi assuré. Les classes, si instances directes de la classe prédéfinie *Class*, sont des objets partageables non modifiables ne posant donc pas de problèmes de cohérence. Cette immuabilité a des conséquences comme d'interdire à une classe de connaître ses sous-classes, ainsi la classe *Object* peut-elle être partagée entre tous les utilisateurs sans problème de cohérence.

Demander un objet, une classe, un descripteur de champ sont des requêtes que les sites s'envoient, il est naturel alors d'encoder ces requêtes comme des objets de classe prédéfinie afin de profiter du même procédé de transfert. Deux classes principales de requêtes existent : *ObjectSendRequest* et *ObjectFetchRequest*. La première transmet à un site une référence sur un objet connu de l'émetteur, la seconde provient d'un site soucieux d'associer une réplique à une sortie et s'adresse au site original hébergeant l'objet. Les réponses aux requêtes passent également par des objets : la classe *ObjectFetchAnswer* ramène le contenu d'un objet tandis que *FailureAnswer* rapporte un code d'erreur. Les requêtes et réponses sont copiables et ont des classes prédéfinies de manière à alléger leur gestion.

La sérialisation d'un objet est donc entièrement contrôlée par sa classe. Ses champs sont successivement sérialisés conformément à leur type : le codage ainsi réalisé est donc très compact puisque le contenu d'un champ est, le plus souvent, sérialisé en le même nombre d'octets que sa représentation physique. Du point de vue de l'implantation, un type est un objet normal de DMeroon qui contient, entre autres, un pointeur vers ses fonctions de sérialisation et de désérialisation.

Le seul type de champ qui pose un problème est la référence à un objet de DMeroon qui se traite, dans le cas général, par l'externalisation de l'objet pointé suivi de la transmission des informations référençant son entrée associée (c'est-à-dire un numéro IP, un numéro de port et un numéro d'objet). Ce schéma purement paresseux impose au récepteur de demander les répliques lui manquant au fil de ses besoins. Toutefois, s'il permet de respecter la topologie des objets partagés, *(i)* il implique néanmoins de nombreux échanges de requêtes, *(ii)* son grain fin dissocie des objets qu'il serait bon de ne pas séparer (la classe et ses descripteurs de champs par exemple).

La désérialisation d'un objet est également contrôlée par sa classe et c'est pourquoi sa classe doit être localement utilisable.

Deux sortes d'améliorations au moins sont envisageables à partir de cet algorithme de base :

- diminuer la taille de la sérialisation et donc coder plus compactement les objets apparaissant souvent ou plus d'une fois,
- pré-envoyer les données dont le récepteur aura certainement besoin afin de lui éviter de les demander ultérieurement.

Le premier but peut être assuré par un codage court des objets les plus utilisés ainsi que par l'utilisation d'un cache (ou d'une pile) mémorisant des objets transmis. Lorsque l'émetteur sait que le récepteur connaît un objet ou une classe, il peut mettre à profit cette information pour pré-envoyer ces objets. Symétriquement, le désérialiseur de DMeroon demande par anticipation certains objets qui seront, vraisemblablement, sous peu, nécessaires.

L'ennui de toutes les gestions de cache est que ceux-ci doivent avoir une taille appropriée mais bornée de manière à ce que l'émetteur ne passe point trop de temps à le scruter. Plutôt donc que d'avoir une politique de gestion de cache générale et intangible mieux vaut procurer au sérialiseur des instructions de gestion de cache spécifiant l'usage que le receveur devra en faire.

Le dernier pas est de faire en sorte que toutes les autres actions de décodage du receveur soient aussi spécifiées comme des instructions pour que le désérialiseur apparaisse comme l'interprète d'un certain langage.

4. Algorithme de transfert en DMeroon

Le désérialiseur de DMeroon est donc un interprète d'octets, disposant d'une pile-cache, et reconnaissant actuellement une quarantaine de commandes. Le désérialiseur s'applique sur un message venant d'une connexion Tcp ; le site émetteur est donc toujours connu.

À chaque connexion Tcp est associé un cache. Les caches ne sont pas partagés. Lorsque l'émetteur insère un objet dans le cache, il engendre aussi les instructions nécessaires à l'insertion de ce même objet dans l'image de ce cache chez le récepteur. Si, par exemple par coopération avec le GC, l'émetteur découvre qu'un objet est inutile, il peut le retirer de son cache et engendrer les instructions nécessaires pour que le récepteur fasse de même. Le cache est donc sous le contrôle programmatique explicite du sérialiseur. Tout comme la table de hachage des sorties, les caches en émission ne sont pas des racines du GC local.

Le langage est formé de commandes suivies d'arguments. Une commande retourne un objet de DMeroon en résultat. Ces objets peuvent être mémorisés en pile ou en cache d'où certaines commandes peuvent les extraire. Les commandes peuvent être enchassées arbitrairement : le résultat d'une sous-commande peut servir d'argument à la commande englobante. La pile n'est pas limitée en taille et croît suivant les besoins. La désérialisation d'un champ, de type référence à un objet de DMeroon, est en fait réalisée par la désérialisation d'un objet normal dont l'adresse sera stockée dans ce champ.

Voici les principales commandes de ce langage.

- *citer objet prédéfini ou implicite* : ces commandes retournent des objets prédéfinis tels que classe ou descripteur de champ. Il est également possible de citer le site courant (récepteur) ou le site émetteur ainsi que quelques constantes universelles comme les booléens Vrai ou Faux.
- *créer objet* : cette commande est suivie d'une référence à une classe (qui doit être locale au receveur) et d'une suite d'entiers naturels correspondant aux tailles des champs indexés de cette classe (s'il y en a). La valeur retournée est précisément cette instance.

- *initialiser un objet* : remplir le contenu de l'objet courant en désérialisant les octets qui suivent. La valeur retournée est l'objet initialisé.
- *rendre copiable/partageable* : ces commandes rendent l'objet courant partageable ou copiable. Elles retournent l'objet courant en valeur.
- *créer référence* : cette commande permet de créer une référence sur un objet (éventuellement distant). Elle est suivie d'un site (numéro IP et port Tcp) et d'un numéro d'objet. En fait, trois instructions spécialisées sont ici confondues qui ont des codages très différents. Les deux premières correspondent à des cas spécialisés très courants.
  - *créer une référence sur l'émetteur* : le site est alors implicitement connu, seul le numéro identifiant l'entrée sur l'émetteur est nécessaire. On transmet également la classe de l'objet à tout le moins sa référence.
  - *créer une référence sur le récepteur* : le site est également implicitement connu puisque c'est le site courant. Seul le numéro identifiant l'entrée sur le récepteur est nécessaire.
  - *créer une référence sur un site qui n'est ni l'émetteur ni le récepteur* : c'est le cas général (mais pas nécessairement le plus courant) qui nécessite de passer l'ensemble des informations ainsi que la classe de l'objet à tout le moins sa référence.
- *lier* : cette commande est suivi d'une sortie et d'un objet quelconque, elle fait en sorte que l'objet devienne une réplique locale de la sortie. Elle retourne l'objet.
- *gérer la pile* : Ce sont les commandes classiques:
  - *empiler un objet* : l'objet courant est empilé ;
  - *dépiler* a pour valeur le sommet de pile ;
  - quelques autres instructions sur la pile à la Forth comme *dupliquer* le sommet de pile, *permuter* le sommet et le sous-sommet, etc.
- *extraire un objet* prend un index et retourne l'objet situé à cet index dans la pile considérée comme un cache ; deux variantes existent selon que l'on compte à partir du sommet de pile ou à partir de la base du cache. Cette dernière variante est à rapprocher de l'instruction suivante :
- *mémoriser un objet* : prend un index et remplace, dans le cache, l'élément de rang correspondant par l'objet courant. Cette instruction permet de gérer le cache c'est-à-dire, insérer ou supprimer des positions afin de ne pas l'encombrer d'objets qui peuvent devenir inutiles.
- *vider complètement la pile/cache.*
- *traiter requête* : elle est suivie d'un objet de la classe *Request* (ou d'une sous-classe telle que *Answer*) qu'il s'agit de traiter.
- *séquence* : elle est suivie de deux objets et retourne le second d'entre eux. Elle permet notamment d'encoder plusieurs requêtes dans un même message.

L'instruction de citation d'objets ci-dessus décrite prend en compte la notion d'objets dotés d'ubiquité. Les classes prédéfinies (*Class*, *Site*, ...) sont immuables et partout les mêmes : elles sont partagées par tous les utilisateurs. Bien qu'implantatoirement statiquement allouées sur chaque site DMeroon, elles sont considérées comme égales. C'est aussi le cas des descripteurs de types de DMeroon qui, quoique possiblement différemment implantés (en taille ou alignement) sur les sites DMeroon, correspondent à un unique concept. Quelques constantes additionnelles, par exemple pour représenter les booléens, ont des codages courts sur un octet.

Voici un exemple de sérialisation d'un vecteur partageable (objet de la classe modifiable *MutableVector*) contenant deux références ; la seconde est sur lui-même tandis que la première pointe sur un entier naturel de taille 4 octets (de la classe immuable et copiable *Nat3*), voir figure 6.

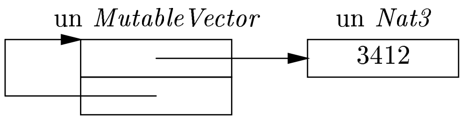

Figure 6. *Exemple d'objets à sérialiser*

Voici trois façons (entre autres) de sérialiser la donnée de la figure 6. Les parenthèses qui apparaissent dans ces programmes n'y sont que pour les rendre plus lisibles en associant mieux commandes et arguments. La première méthode transporte le miminum et se contente d'établir un pointeur distant. On suppose que n est le numéro attribué à l'entrée associée au vecteur.

(référenceSurÉmetteur n (citerClasse *MutableVector*))

La seconde méthode favorise l'enchâssement de commandes et n'use aucunement de la pile. Par contre, elle laisse au récepteur le soin de raccourcir les pointeurs distants associés à une réplique locale.

(lier (référenceSurÉmetteur n (citerClasse *MutableVector*))
    (rendrePartageable (initialiser (créer (citerClasse *MutableVector*) 2)
        (rendreCopiable (initialiser (créer citerClasse *Nat3*) 3412))
        (référenceSurÉmetteur n (citerClasse *MutableVector*)))))

La troisième méthode linéarise les créations d'objets puis les remplissages des dits objets, elle use temporairement de quatre positions dans la pile.

(séquence empiler (citerClasse *MutableVector*)
(séquence empiler (créer sommetPile(0) 2)
(séquence empiler (créer citerClasse *Nat3*)
(séquence empiler (référenceSurÉmetteur n sommetPile(-2))
(séquence (lier sommetPile(0) sommetPile(-2))
(séquence (initialiser sommetPile(-1) 3412)
(séquence (rendreCopiable sommetPile(-1))
(séquence (initialiser sommetPile(-2) sommetPile(-1) sommetPile(-2))
(séquence (rendrePartageable sommetPile(-2))
(séquence dépiler
(séquence dépiler
(séquence permuter dépiler))))))))))))

La partie la plus complexe concerne la sérialisation d'un pointeur sur un objet de DMeroon. Si l'objet pointé est doué d'ubiquité, il est directement encodé. S'il apparaît dans le cache, alors on se contente d'engendrer l'instruction qui l'en extraiera sur le récepteur. S'il est d'une classe prédéfinie, il est encodé à l'aide de la fonction spécifique de sérialisation que toute classe prédéfinie inclut. En effet, un objet telle qu'une classe référence des descripteurs de champs qui à leur tour référencent la classe qui les a introduits. Pour briser ce cycle, un sérialiseur spécialisé de classe est utilisé. Si l'objet est d'une classe non prédéfinie mais connue par le récepteur (par exemple, parce qu'elle apparaît dans le cache), alors on peut l'encoder directement mais en prenant garde de l'insérer dans le cache au cas où il se référencerait lui-même.

Comme on a pu le lire, l'invariant majeur à respecter est que l'on ne peut désérialiser un objet dont la classe n'est pas localement connue. Toutefois, les classes se propagent assez rapidement et il est suboptimal de n'en point profiter. Nous avons introduit récemment une nouvelle commande : *tenter* (non sans rapport avec l'`unwind-protect` de Lisp). L'instruction (tenter n *objet1 objet2*) tente de désérialiser l'*objet1* dont la sérialisation tient en n octets. Si aucune erreur n'arrive pendant la désérialisation, c'est cet objet qui est retourné en valeur et le second objet est lu puis ignoré. Si une erreur survient alors la désérialisation reprend avec l'*objet2* qui sera retourné en valeur d'où l'importance de savoir où reprendre. L'usage le plus fréquent de cette commande est le suivant:

(tenter n (initialiser (créer ...) ...) (référence ...))

Ce tour permet d'envoyer par anticipation le contenu d'un objet dont la classe n'est peut-être pas connue du récepteur. Dans ce cas l'initialisation de l'objet sera erronée donc abandonnée au profit d'une simple référence lointaine. Si, par contre, la classe était utilisable alors le récepteur bénéficie immédiatement de l'objet et gagne ainsi au moins un échange de message.

Pour résumer, ce langage permet d'implanter de nombreuses politiques de transfert de valeurs : paresseuse, avec anticipation d'envoi, avec mémorisation et synchronisation de caches, etc. C'est au compilateur de choisir et de mettre en œuvre sa politique de sérialisation à interprète de désérialisation constant.

5. Amorçage

Lorsque deux sites DMeroon sont reliés par une connexion Tcp, ils s'échangent les objets de la classe *Site* qui les décrivent. Les objets de cette classe prédéfinie sont modifiables et donc nécessitent de passer par le mécanisme des pointeurs distants pour être partagés de façon cohérente. Or une sortie ne contient que le numéro de l'entrée distante et un pointeur sur l'objet de la classe *Site* désignant le site contenant cette entrée distante, voir figure 7. Un site distant est donc une structure cyclique complexe qui est la première à être créée lorsqu'une connexion Tcp est ouverte. Ce problème d'automorçage est réglé, lors de l'établissement de la connexion, par une commande appropriée de transfert d'instances de la classe *Site*. Cette méthode restreint actuellement les sites à être des instances directes de la classe *Site*.

6. Conclusions

Après avoir présenté certains détails d'implantation du mécanisme des pointeurs distants et de leur gestion, ainsi que les mécanismes de diffusion paresseuse des classes nouvellement créées, cet article a décrit les grandes lignes d'un procédé de sérialisation/désérialisation fondé sur un langage d'échange. Le désérialiseur n'est qu'un interprète de ce langage dont le compilateur est le sérialiseur. Il est ainsi possible *(i)* d'adapter le comportement de la sérialisation aux caractéristiques des valeurs à sérialiser, *(ii)* d'accumuler rapidement des expériences à interprète constant, *(iii)* de faire évoluer le protocole d'échange par ajout de nouvelles instructions.

L'interprète de désérialisation de DMeroon a une taille de 16Ko tandis que l'actuel sérialiseur mesure 6Ko. Ces chiffres sont indépendants du nombre de classes existantes ou futures dont la description occupe 13Ko. À titre de comparaison et sur la même machine, `rpcgen`, appliqué aux quelque 80 classes prédéfinies de DMeroon, engendre un code pesant 19 Ko auxquels il faut ajouter les morceaux nécessaires de la bibliothèque C (45Ko sur mon Linux favori). Le (dé)sérialisateur a donc une taille largement compatible des actuelles tailles de cache d'instructions. De plus cette solution logicielle est portable, sans dépendance aucune vis-à-vis du système d'exploitation, et reste compatible d'un GC recyclant, par exemple, les classes inutilisées.

La sérialisation mène à un train d'octets qui peut également servir à archiver des objets dans des fichiers de format éminemment portable. On peut aussi envisager de transmettre ces objets encodés (des *oblets* ?) dans des pages HTML.

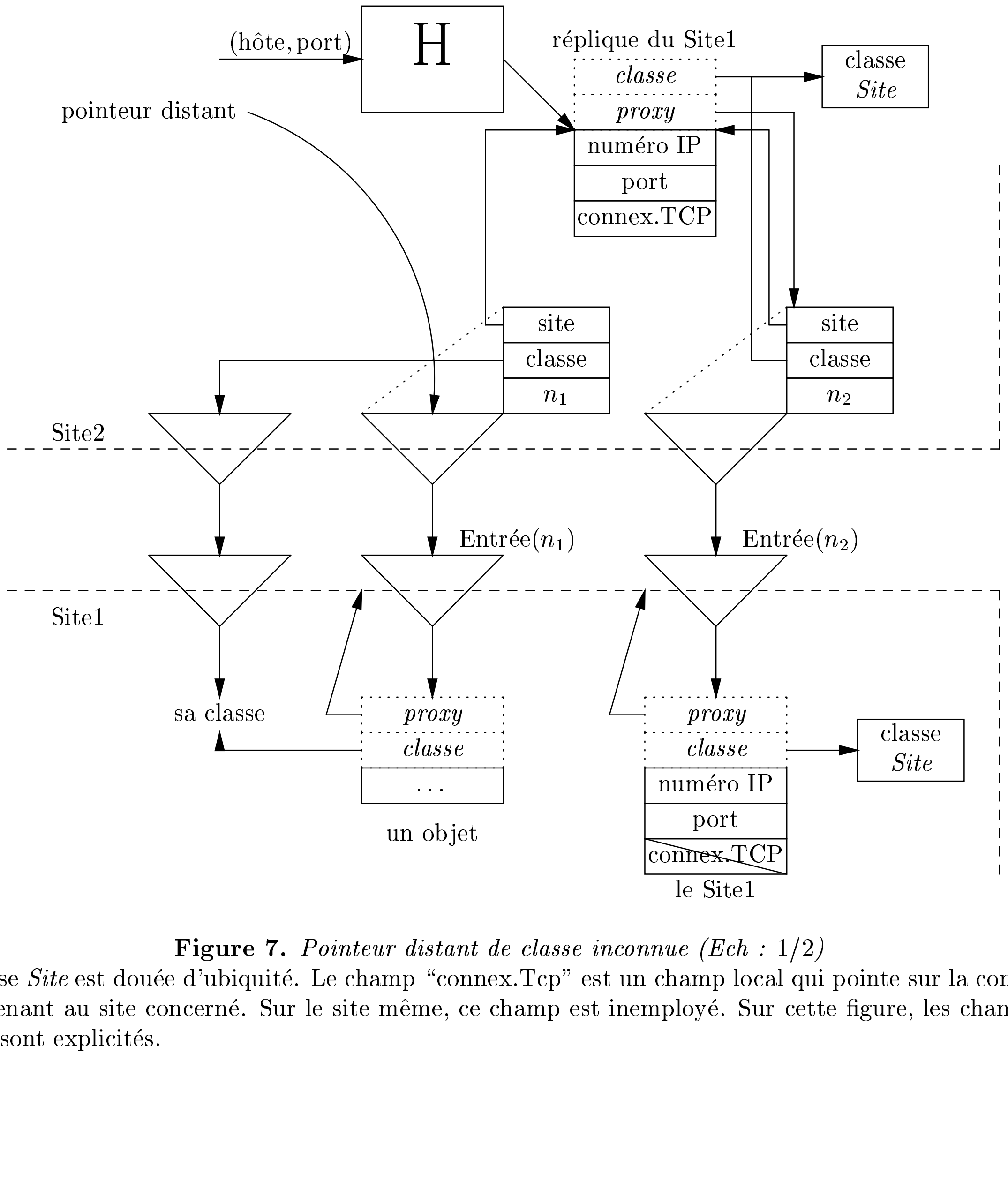

Figure 7. *Pointeur distant de classe inconnue (Ech :* 1/2*)*
La classe *Site* est douée d'ubiquité. Le champ "connex.Tcp" est un champ local qui pointe sur la connexion Tcp menant au site concerné. Sur le site même, ce champ est inemployé. Sur cette figure, les champs des sorties sont explicités.

Que ce train d'octets soit exécutable pose les problèmes usuels de sécurité. Un sérialisateur pervers peut référencer et modifier des objets du récepteur. DMEROON apporte deux réponses: les numéros des entrées qui identifient les objets sont des numéros difficilement falsifiables car ils comportent une clef aléatoire de 16 bits), d'autre part, l'ouverture d'une connexion entre deux sites DMEROON requiert un mot de passe.

Une extension envisageable, souhaitable et utile est de coupler DMEROON au noyau de Linux à la manière de [Bartoli 97]. Ainsi le train d'octets produits (ou décodés) est-il directement engendré dans (ou lus depuis) les files d'entrée/sortie du système d'exploitation sans intermédiaire inutile.

Le système DMEROON est documenté en :

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

7. BIBLIOGRAPHIE

[Baker 93] Henry G Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4):2–27, October 1993.

[Bartoli 97] A Bartoli. A novel approach to marshalling. *Software — Practice and Experience*, 27(1):63–86, January 1997.

[Boehm 88] Hans J Boehm et M Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9), September 1988.

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

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

[Lecouffe 79] Pierre Lecouffe. Communication et mémorisation en lisp. Dans *GROPLAN*, numéro 9, pp. 37–46. AFCET groupe programmation et langages, 1979. Conférence “Panorama des langages d'aujourd'hui”, Cargese (France), 14-22 mai 1979.

[Moreau 97] Luc Moreau, David DeRoure, et Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. Dans *Third International Europar Conference (EURO-PAR'97)*, Lecture Notes in Computer Science, Passau, Germany, August 1997. Springer-Verlag.

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

[Protić 95] J Protić, M Tomašević, et V Milutinović. A survey of distributed shared memory systems. Dans *Proc. 28th annual Hawaii International Conference on System Sciences*, vol. I (architecture), pp. 74–84, 1995.

[Queinnec 93] Christian Queinnec et David De Roure. Design of a concurrent and distributed language. Dans Robert H Halstead Jr et Takayasu Ito, éditeurs, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, vol. Lecture Notes in Computer Science 748, pp. 234–259, Boston (Massachussetts USA), October 1993.

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

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

[Shapiro 92] Marc Shapiro, Peter Dickman, et David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), nov 1992. Also available as Broadcast Technical Report #1.

[Steele, Jr. 90] Guy L. Steele, Jr. *Common Lisp, the Language.* Digital Press, Burlington MA (USA), 2nd édition, 1990.

[xdr] Rfc 1014: external data representation standard: Protocol specification. Rapport technique, ARPA Network Information Center.

∞

Biographie

Élevé, il y a plus de deux décennies, dans un giron fortement Lispien, C.Queinnec s'intéresse depuis quelques années aux langages répartis et plus particulièrement à la gestion automatique de mémoire (GC), à la cohérence d'espaces d'adressage disjoints ainsi qu'à la programmation énergétique. Présentement professeur à Paris 6, il continue d'apprécier les langages applicatifs, le chou-fleur au gratin et l'écriture en (France-)musique.

Pour en savoir plus:

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